DESIGN OF TWINE – RISC

By

DINESH RAO B.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

MARCH, 1993

DESIGN OF TWINE-RISC

A thesis submitted
in partial fulfillment
of the requirements
for the degree of

Master of Technology

by

Dinesh Rao B

to the

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

February, 1993

CSE-1993-M-RAO-D

CERTIFICATE

This is to certify that the work contained in the thesis titled. **DESIGN OF Twine-RISC**, was carried out under my supervision by **Dinesh Rao B** and it has not been submitted elsewhere for a degree.

[illegible]

Dr. Rajat Moona
Asst. Professor
Dept. of Comp. Sc. and Engg.
I.I.T., Kanpur

Abstract

RISC machines, derived from the conventional von Neumann architecture are easy to design and have tremendous computing power. However, the stored program concept and the centralized control of these computers have led to inter-dependence of instructions. This leads to inefficient execution of parallel programs.

Twine-RISC combines the ideas of von Neumann and Dataflow architectures and eliminates the drawbacks of RISC machines in parallel execution of programs. In this thesis, we have proposed a simple design for *Twine-RISC*. The proposed design of two stream *Twine-RISC* processor consists of less than 40k transistors. A novel design for a *queue* has also been implemented in VLSI. In our *queue*, Read and Write operations take less than 1ns.

ACKNOWLEDGEMENT

I am very grateful to my guide, Dr RAJAT MOONA, for offering constant guidance throughout my thesis work.

I thank Shantakka, Harish, Galagali. Sudheer and my other friends for helping me when I was down with Jaundice. I also thank my friends for making my stay at I. I. T. K very pleasurable.

DINESH RAO B.

19 - 03 - 93.

Contents

List of Figures

Chapter 1

Introduction

RISC architectures[3] have been derived from the conventional von-Neumann architecture. This architecture is widely used in modern commercial computers of these days. The RISC instructions are simple, regular and are usually based on three operands. However, in RISC, the inter-dependence of instructions due to the stored program concept of von Neumann computers have proved to be a major bottleneck in parallel execution of programs.

Dataflow architectures[4] offer a possible solution for efficiently exploiting concurrency of computation on a large scale. The computing nodes are fired when the data arrives and the execution of instructions may not be in the sequence in which they are stored in the memory of a computer[6]. However, no architecture supports efficient execution of dataflow programs.

Nikhil and Arvind[1] proposed P-RISC, which combines the ideas of both von-Neumann and dataflow computing. In P-RISC, the program counter(PC), found in von-Neumann computers is eliminated and multiple threads of computation is achieved through the execution of tokens, a concept borrowed from dataflow computing[4,6,14]. The RISC feature of pipelined instruction execution is also effectively utilized. However, in a single processor, multiple threads cannot be simultaneously executed.

Moona, Nandy and Rajaraman [2] have proposed a novel architecture called *Twine-RISC* which supports multiple threads execution in a single processor. *Twine-RISC* has eliminated many drawbacks which are existing in P-RISC and efficiently exploits (fine- grain) parallelism which is inherent in most programs. In this thesis, we propose a simple design for *Twine-RISC*.

The two stream *Twine-RISC* consists of less than 40K transistors. The details of the transistor count has been given in chapter 4. The design can be easily modified to accommodate more Twine-RISC streams. All blocks in *Twine-RISC* pipeline operate asynchronously with handshaking modules between each pair of blocks taking care of data overruns. A novel design for *queuecell* has also been implemented in VLSI.

The rest of the chapters are organized as follows. In chapter 2, we survey RISC, dataflow and a combination of RISC and dataflow architectures. In chapter 3, we elaborate on *Twine-RISC* architecture. In chapter 4, we propose the hardware realization of *Twine-RISC* and in chapter 5, we conclude with the results and a brief note on the future work to be done in this field. Appendix A explains the detailed instruction set of *Twine-RISC*. Appendix B gives the coding of the *Twine-RISC* instruction set.

Chapter 2

RISC and Dataflow Computers

In this chapter we discuss the origin and the development of the RISC and dataflow machines. We also discuss about the attempts towards fusion of dataflow and von-Neumann ideas to develop hybrid processors for parallel execution. Section 2.1 discusses the evolution of RISC from CISC. Section 2.2 describes the common RISC features found in many machines. In section 2.3, we discuss variations in the RISC designs. Memory latency is an important factor which determines the maximum speed of instruction execution in any RISC machine. We discuss about the effects of Memory latency on the processor performance in section 2.4. In section 2.5, we discuss the dataflow architecture and in section 2.6, we discuss the attempts made to synthesize a new architecture by fusing the ideas of von Neumann and dataflow architectures. We discuss about P-RISC[1] and *Twine-RISC* in section 2.7.

2.1 Introduction to RISC

RISC (Reduced Instruction Set Computer) designers advocate a simple set of instructions which can be easily executed in pipelines. Before the advent of RISC, there was a notion that smaller code size, more number of instructions and more addressing nodes would improve the performance of the computer. However, researchers of RISC-1[3] observed that in majority of applications, a very small set of instructions(30-40) were frequently used out of a large set of instructions (about 250-300) in a CISC (Complete Instruction Set Computer). Also, the large number could be implemented from this basic set of instructions.

A RISC machine had evolved because of the following design principles

- Functions should be kept simple unless there is a very good reason to do so. Addition of an extra instruction should justify its inclusion.
- Microinstructions should not be considered to execute faster than the simple instructions. The speed difference between control memory and other memories is not large in the current scenario.
- Simple decoding and pipelined execution are more important than program size. Increasing the number of instructions increases the complexity of the decoding and the controlling circuitry, thereby reducing the execution speed. Smaller program size need not always imply faster execution.
- One should simplify the instructions rather than complicating them.

2.2 Common RISC features

We can see the following common features in some actual RISC machines like the 801 from the IBM research, RISC-1 and RISC-2 from Berkeley and MIPS from the Stanford university.

- Operations are register-register with only LOAD/STORE instructions accessing memory.
- The operation and addressing modes are reduced to a large extent. Typically, there are 30-40 instructions and 3-4 addressing modes.
- Instruction formats are simple and do not cross word boundaries.
- RISC branches avoid pipeline penalties. Pipelines should normally be flushed when a branch instruction takes place. The RISC solution is to change jumps such that they take place only after the next instruction is executed. This is called the *delayed branch.* The machine language code is suitably shuffled for execution by the compilers.

2.3 RISC variations

2.3.1 Register Windows

Many registers were provided by the designers of RISC-1 and RISC-2 to keep all the local scalar variables and all the parameters of the current procedure in the registers. They also provided many register sets, or windows, of registers so that registers need not be saved on every procedure call and restored on every return. Instead of copying the parameters from one window to the other on each call, windows are overlapped so that some registers are part of two windows. By putting the parameters into overlapping registers, operands are automatically passed.

2.3.2 Pipelines

All RISCs use pipelined execution, but the length of the pipeline varies. A balance has to be found between the four parts of a RISC instruction execution, namely

- Instruction fetch
- Register read
- Arithmetic / Logic operation
- Register write.

The 801 assumes that each part takes the same amount of time and hence uses a four-stage pipeline. RISC-1 and RISC-2 assume that instruction fetch was equal to register read plus arithmetic/logic operations and thus have a three-stage pipeline.

2.4 Memory Latency

It is the time taken between a memory request and its corresponding response. This is the main factor that determines the maximum instruction speed. von Neumann processors are likely to *idle* during long memory references. Caches reduce memory latency. Increasing the number of registers and using multiport memories which are capable of receiving multiple overlapped requests are other ways of reducing memory latency. Some Dataflow processors use I-structure memories[5] which eliminate the problem of memory latency by allowing split-phase transactions.

2.5 Dataflow machines

Dataflow architectures give a possible solution for efficiently exploiting concurrency of computation on a large scale. An instruction in a dataflow machine is executed when the operands of its dataflow graph have arrived. Dataflow machines do not have program location counters and there is no concept of control flow. A consequence of data-activated instruction execution is that many instructions of a dataflow program may be available for execution at once. Thus, concurrency is a natural consequence of the dataflow concept.

2.5.1 Dataflow graphs

A dataflow program graph is a directed graph where each node is an operator with lines connecting an output port of one operator to an input port of another, provided that no two outputs are connected to the same input (Fig 2.1). Values are carried by tokens which move along the line between the operators. Execution of a program graph is *data driven* in that an operator executes by absorbing exactly one token from each input, computing results and producing exactly one result token for each active output.

The dataflow graph specifies the partial order of execution of instructions and hence it provides opportunity to execute instructions in parallel. This also allows implicit sequencing of instructions and allows them to execute in parallel if there is no dependency.

2.5.2 The basic execution mechanism in Dataflow machines

The basic instruction execution mechanism used in several current dataflow projects is shown in Fig 2.2. The dataflow program describing the computation to be performed is held as a collection of activity templates in the activity store. Each activity template has a unique address which is entered in the instruction queue unit(a FIFO) when the instruction is ready for execution. The fetch unit takes an instruction address from the instruction queue and reads the activity template from the activity store, forms it into an operation packet and stores it in the operation unit. The operation unit performs the operations specified in the operation code on the operand values and generates one result packet for each destination field of the operation packet. The update unit receives result packets and enters

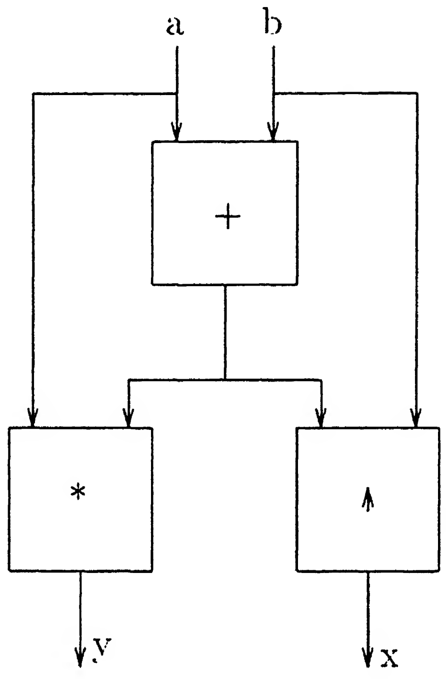


Figure 2.1: A Dataflow Graph

the values they carry into operand fields of activity template as specified by their destination fields. This mechanism is thus a circular pipeline.

Many architectures on dataflow mechanism have been proposed. Some have been implemented in experimental machines[6].

2.5.3 The I-structure Memory

I-structures[5] are a way of introducing a limited notion of state into dataflow graphs, without compromising parallelism or determinancy. I-structures reside in a global memory. A *producer* dataflow graph writes into an I-structure location while several other *consumer* dataflow graphs read that location. I-structure semantics require that consumer should wait until values becomes available in its location.

The I-structure memory also alleviates the problem of memory latency by allowing multiple *split-phase* transactions.

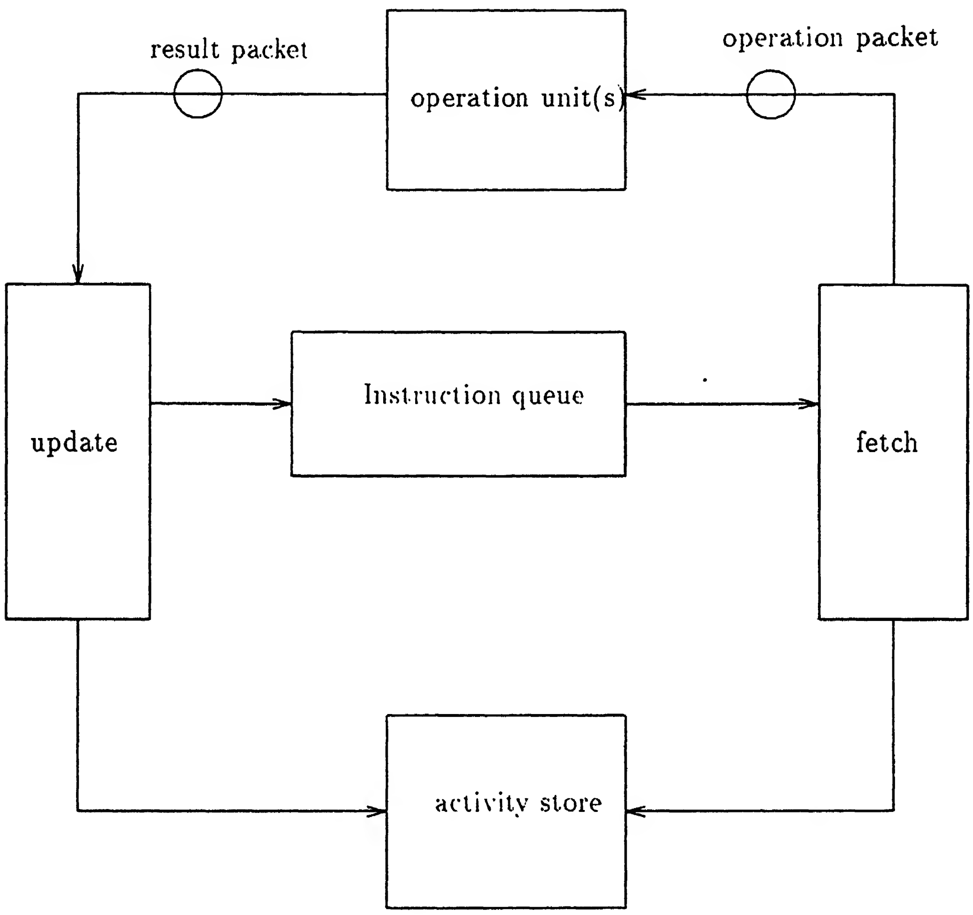


Figure 2.2: Basic execution mechanism in Dataflow Architectures

2.6 Incorporating Dataflow ideas in von-Neumann processors

The approaches taken by the conventional and dataflow architectures are contrasted in relation to each other in the issues of memory latency, synchronization and distribution costs in a multicomputer. Each approach fits well for specific applications and it is possible to have a framework in which the two execution models can be mixed to suit the situation. Existing programs must run without much change and improvements could be made by using new features to tolerate memory latency and to exploit more parallelism.

Conventional multiprocessors connect a set of processors to a global memory through a communication network. Computation on different processors can be carried out independently. whereas processors communicate through the shared memory and appropriate synchronization is accomplished through program primitives. There are two sets of problems with conventional multiprocessors

- When a program written for a single processor is being converted to run on many processors, overheads such as detection of parallelism, separating local and global variables, inserting appropriate synchronization commands fall on the programmer.
- Problems due to the execution model which is based on a sequential control, such as cache inconsistency, pipeline delays, busy waits etc.

Let us contrast the conventional and dataflow approaches in relation to these problems. The dataflow model has no single locus of control, i.e., instructions like add, subtract, multiply etc are executed upon the arrival of data, thus solving the first set. The token model[11, 12] solves the second set of problems. One can also solve the second set of problems if one can recognize an event, such as global memory access. which can take indeterminate time and have a fast context switching capability. Thus, by adopting programming discipline and by capturing the essentials of dataflow principles, one can solve the two sets of problems with almost the same style of processors we have today.

The dataflow model does promise to extract the last ounce of parallelism in an algorithm specification. However, the dataflow model also calls for a radical departure in both programming methodologies and in architecture. Normally, such radical shifts are very hard because of the presence of large body of existing software and the high cost of developing a new architecture.

Let us highlight the following points.

- The new architecture should not be radically different from those in existence.
- Existing software should run without substantial modifications or loss in efficiency.

2.7 P-RISC and Twine-RISC

In P-RISC[1], a recent trend towards a synthesis of dataflow and von-Neumann architectures has been utilized. It has a RISC like instruction set and uses an efficient RISC pipeline. It has eliminated the use of PC (Program Counter). P-RISC has memory for code and frames. The location of the next instruction that could be executed is given by the Continuation process descriptor (FP, IP). FP is the Frame Pointer which points the current frame in the frame memory and IP is the instruction pointer indicating the instruction to be executed in the code memory. A tag (FP, IP) is generated at the end of each instruction and is stored in the token store. The loads are split-phase[1] transactions so that the path to the memory is not occupied during the entire period.

Twine-RISC [2] is a modification of P-RISC which efficiently exploits the fine-grained parallelism existing in the programs by allowing multiple threads to execute simultaneously without a multiprocessor interconnection. In *Twine-RISC*, a Continuation Token (FP, IP) is generated only when a new *thread* is to be executed. A *thread* is a sequence of dependent instructions. The Instruction Fetch Unit(IFU) checks if the current instruction is a non-jump type instruction and increments the IP (effectively incrementing the PC). A new Continuation Token is generated by the Execution Unit (EXU) only if the instruction is not of Arithmetic/ logic type. type. This eliminates the generation of unnecessary tokens. If more than one token is present in the token store, the multiple Twine -RISC Streams (TRSs) of the *Twine-RISC* execute simultaneously, thereby injecting greater parallelism in instruction execution. Multiple load requests can be issued to the global memory and

Twine-RISC can accept responses in a different order in which requests were issued. The processor is *not idle* until the response to the Load arrives from the global memory.

2.8 Conclusion

Dataflow and RISC architectures have their own merits and demerits. *Twine-RISC* is a processor which effectively fuses the advantages of von-Neumann and dataflow architectures. Multiple threads can be efficiently executed in *Twine-RISC* The next chapter describes the processor architecture of the *Twine-RISC*.

Chapter 3

Twine-RISC ARCHITECTURE

3.1 Introduction

In this chapter, we discuss the processor architecture of *Twine-RISC*. The multiple RISC pipeline in *Twine-RISC* effectively exploits the instruction level parallelism and simultaneous execution of multiple threads. The *Twine-RISC* supports *split-phase* transactions between the global memory and the processor through the message processor. Fig. 3.1 illustrates the processor architecture of the *Twine-RISC*. In the subsequent sections we discuss the major functions of each block of *Twine-RISC* viz., the Operand Memory(OM), the Code Memory(CM), the Token Queue(TQ), the Data Queue(DQ), the Sequencer, the Message Processor(MP) and the TRS. The various stages in the TRS are the Instruction Fetch Unit(IFU) . the Operand Fetch Unit(OFU), the Execution Unit(EXU), the Result Store Unit(RSU). All the units of the TRS operate asynchronously and a handshaking unit is present between each pair of interfaced blocks of a TRS. Handshaking units are also placed between the EXU and the Sequencer. There are two TRSs in the *Twine- RISC* viz., TRS#1 and TRS#2.

3.2 Operand Memory (OM)

The OM concept in *Twine-RISC* is similar to the register file of conventional RISC processors. It consists of 64 words, each word of 32 bit length. The OM is shared by the multiple TRSs. The OM is a 3-port memory structure having two read ports and one write port. For read and write operations,

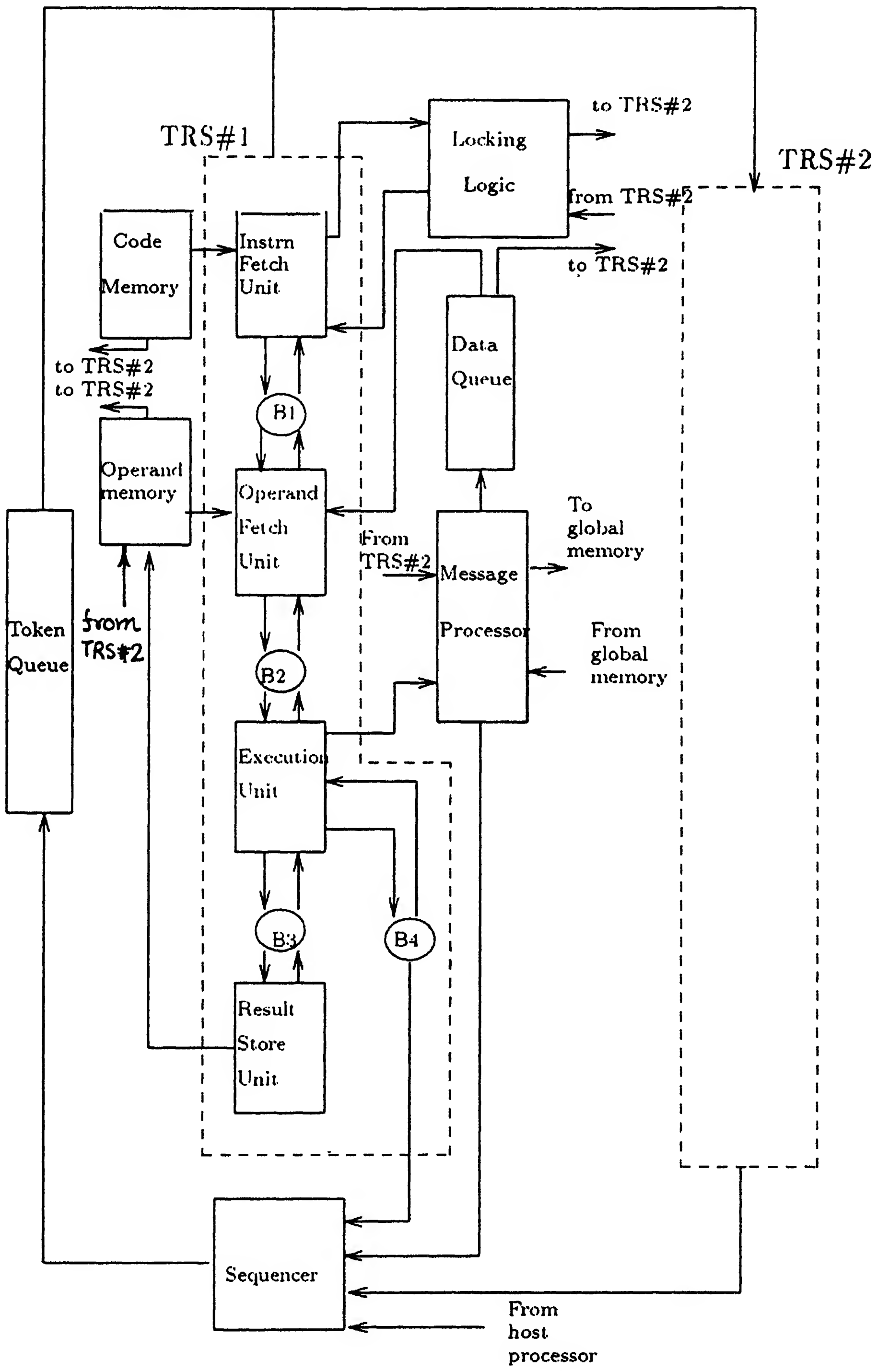


Figure 3.1: Architecture of Twine-RISC

the clock period is split into two phases phi-1 and phi-2, the first phase being utilized by TRS#1 and the second by TRS#2. The read ports are utilized by the OFU's of the TRSs and the write port is used by the RSU's.

3.3 Code Memory (CM)

The CM is common to all TRSs and is positioned outside the chip. It holds the instructions and is read-only for the TRSs. A separate host processor is used to initialize the CM. The design for the CM is trivial and is not included in this thesis.

3.4 Token Queue (TQ)

The continuation tokens for the TRSs are stored in the TQ. A continuation token consists of two pointers, namely the frame pointer (FP) and the instruction pointer (IP). The IP points to the position of the instruction to be executed in the CM and the FP is a base pointer to the data in the OM for a code block. Multiple active invocations for the same code block are possible by the use of frame relative addressing. A continuation token can be utilized by any TRS. The TQ is initially loaded by the *host processor* through the Sequencer and subsequently, the continuation tokens are supplied by the TRSs. Host can also insert tokens later during the program run to make *Twine-RISC* execute threads asynchronously and thereby handle asynchronous events.

3.5 Sequencer

The continuation tokens generated in the system are stored in the TQ after passing through the Sequencer. The two streams of *Twine-RISC* may generate more than one continuation token during a clock cycle. The continuation tokens are also generated by the MP and the *host processor*. The sequencer serializes these tokens and sends them one after the other to the TQ. All the tokens present in the TQ are independent of one another and the sequence in which these tokens are stored in the TQ is irrelevant.

3.6 Data Queue (DQ)

It is similar to TQ and is used to store the data coming from the global memory in response to LOAD/LOADX through the Message Processor. When a memory operation like LOAD/ LOADX is encountered. a value. continuation token and destination register is returned by the Message Processor. These data are written into the DQ and a thread to execute the instruction RESM is added to the TQ. When RESM is executed, data is finally moved from the DQ to the OM and the thread is reintiatated.

3.7 Message processor (MP)

MP takes care of the movement of the data between the TRSs and the global memory. In case of a read request. the MP receives a response from the global memory controller containing a value. Operand Memory address and continuation token. The MP writes data into DQ and generates a continuation token (0,0) to be stored in the TQ. The MP also directs the LOAD/STORE requests to the global memory. The EXU of the RISC pipeline sends 78 bits data to the MP consisting of 1-bit Read. 32-bit address, 32-bit IP, 6-bit Destination Register (DR) and 1-bit request. The MP receives a similar message (77-bits) from the global memory controller without the request line and sends it to the DQ.

3.8 Instruction Fetch Unit (IFU)

Initially, when the restart(RST) line is activated, IFU fetches a new token from the TQ and fetches the corresponding instruction from the CM. For the subsequent instructions. it determines whether the next instruction is to be fetched from the next CM location or not. In case of branch instructions, the IFU fetches a continuation token from the TQ and starts a new thread. Two MJOIN instructions should not execute simultaneously as they may may write to the same OM concurrently. To prevent a race between MJOIN instructions. the IFU detects the MJOIN instruction and sets the *lock* line. The *locking logic* associated with the IFUs prevents simultaneous execution of MJOIN instructions.

3.9 Operand Fetch Unit (OFU)

This TRS block recognizes instructions partially by decoding three bits of opcode and decides the number of operands to be fetched from the OM. This unit also detects the RESM instruction and if so, fetches operands from the DQ. It then routes the instruction, operands, FP and IP to the EXU.

3.10 Execution Unit (EXU)

It is similar to the ALU of a conventional processor. It also prepares continuation tokens (FP,IP) for branch and other special instructions for thread initiation and forwards it to the Sequencer through the buffer B3. After executing the arithmetic and logical instructions, it sends the result and the address of the destination register to the RSU. Requests for memory read/write are sent to the MP.

3.11 Result Store Unit (RSU)

This is the only stage the that can write to the Operand Memory(OM). It writes the value of the result generated by the EXU in the destination register . It also releases the MJOIN *lock* line set by the IFU.

3.12 Conclusion

The architecture of *Twine-RISC* has a simple pipeline structure and a modular design. This helps us in designing each of the blocks in a systematic way with minimum overheads. The next chapter describes the hardware design of *Twine-RISC*.

Chapter 4

Design Of Twine-RISC

In this chapter we describe the design of the *Twine-RISC* processor. There are two TRSs in the *Twine-RISC* viz., TRS#1 and TRS#2 (Fig 3.1). We describe the design of a TRS and its associated modules in this chapter. In the first section, we describe the design of the handshaking module which is present between each pair of blocks in the TRS pipeline(Fig 3.1). The subsequent sections describe the design of the other blocks of the *Twine-RISC*, viz., the IFU, The EXU, the OFU, the RSU, the CT, the OM, the DQ, the TQ, the MP and the Sequencer.

4.1 The Handshaking Module

The function of this module is to guide the transfer of data from one block of the TRS pipeline to another such that both blocks can run independently. Four such modules have been used in each of the TRS pipeline(Fig 3.1). Length of the d-latch is suitably adjusted for each module. Fig 4.1 illustrates the block diagram of such a module. The operation of the handshake module is as follows.

A sending block informs the handshake module that it has data available, while the receiving block has informed the handshake module that it is not using the data available at it. The handshake module now stores the valid data in the latch and acknowledges the receipt. The second step of the module is to tell the receiving block that it has data ready after it has informed the handshaking module that it is ready to accept new data. The sending block now tells the handshake process that its data is invalid. The cycle is finished when the receiving block indicates that it has finished

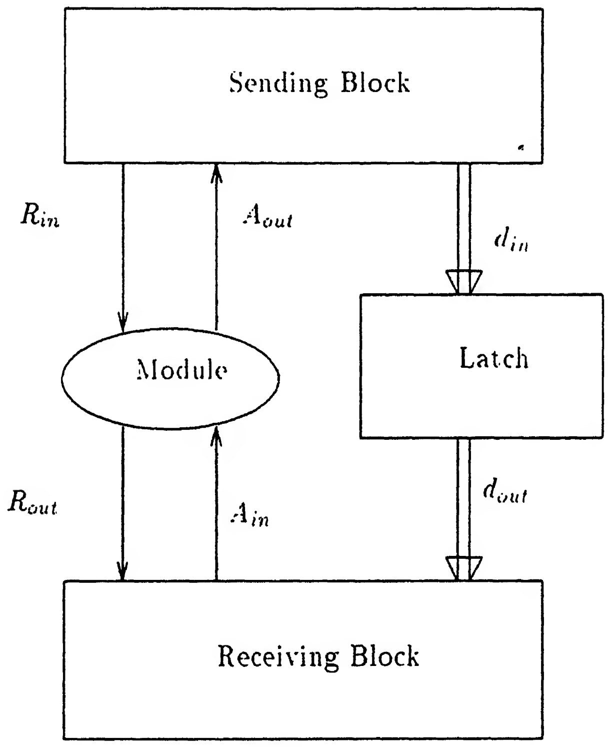
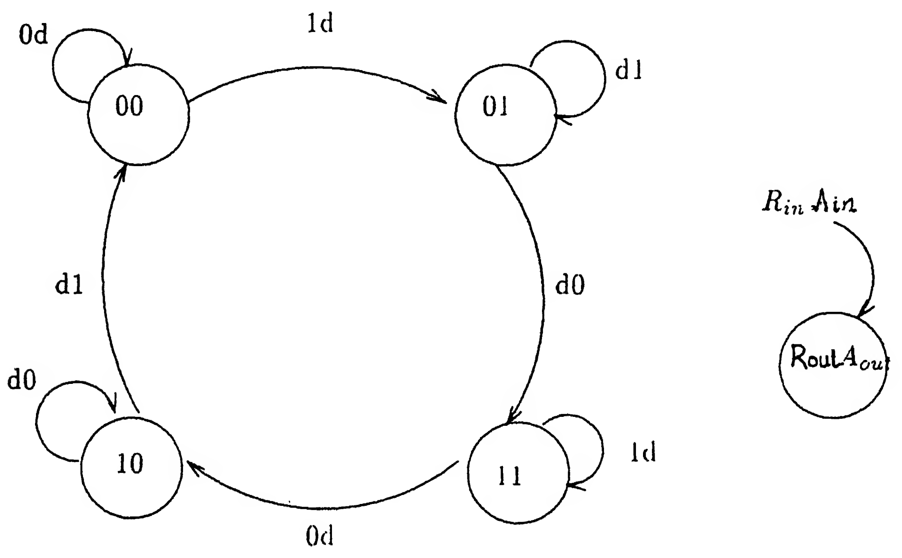


Figure 4.1: Block Diagram of the Handshake Module



d = don't care

Figure 4.2: State Diagram for Handshake module

processing and does not need the data held in the latch any more. R_{in} is the request coming into the handshake module and R_{out} is the request from the module to the receiving block. A_{in} is the acknowledgement coming from the receiving block and A_{out} is the acknowledgement to the sending block.

In the initial state, A_{out} is equal to 0 because no data has been received by the handshake module and R_{out} is equal to 0 indicating that the handshake module has no new data available for the receiving block.

The next state is reached by raising R_{in} by sending block, indicating that it has valid data at its outputs. The handshake module stores the data in the latch and acknowledges by raising A_{out}.

Once the receiving block passes the message that it is ready to accept new data by lowering A_{in}, the handshaking module indicates the receiving process that it can start operating on the new data by raising R_{out}.

The next transition is caused by resetting of R_{in} indicating that the output data is no longer valid data. A_{out} is set to 0 indicating that the handshake module is awaiting new data.

Handshake module is turned into its first state after receiving module has acknowledged receipt and processing of data by raising A_{in}.

From Fig 4.2, we can see that during state (R_{out} =1, A_{out} =1) both blocks can concurrently execute. The *init* signal is set to 0 only at startup time, to initialize to the correct state and is further held at 1. Fig 4.3 illustrates the Karnaugh map for the logic formulas of R_{out} and A_{out}.

From this Karnaugh Map we can derive the minimal logic formulas including the *init* signal. Fig 4.4 gives the logic diagram for the handshake module. The width of the latch[7] should be suitably adjusted for different buffers.

4.2 The Instruction Fetch Unit

The Fig 4.5 illustrates the block diagram of the IFU. 38 input lines are accepted from the TQ (FP(6),IP(32)). 24 input lines from the CM, a *grant* signal from the Locking Logic and a handshaking signal(A_{out1}) from the buffer *B1* . The output lines consist of 32-bit address line to the CM, a request line to the TQ(TQR), a *lock* line (L1) to the Locking Logic, 62 lines to *B1* (6-bit FP, 32-bit IP, 24-bit instruction) and a handshaking signal R_{in0} to the buffer *B1*. The clock cycle is divided into 4 phases viz., phi-1, phi-2, phi-3 and phi-4. Phi-1 and phi-4 are used for handshaking. Phi-2 is used for loading a new token from TQ or incrementing the IP and phi-3 is used for fetching instruction from the CM.

The IFU initially makes sure that it can proceed normally by lowering the signal R_{in0} to *B1*(handshaking buffer between IFU and OFU). If A_{out0} turns low, the IFU proceeds with its normal instruction fetch (Fig 4.6). The d-FF(d flip-flop) keeps track of the state of the IFU (Q = 1 implies normal operation). When the IFU has completed its operation, R_{in0} is asserted during phi-4 to inform *B1* that data is ready in the IFU.

The *Load Logic* Fig 4.7 checks if a new token is to be fetched from the TQ. Bit IR_{23} in the 24-bit*Instruction Latch* (IR) indicates whether the previous instruction was jump/memory/special type (IR_{23} = 1) or Arithmetic/Logic type (IR_{23} = 0). During Phi-2, if IR_{23} = 1, a new token is loaded into FP (6-bit d-latch) and IP (32-bit Incrementing Register) from the TQ. Otherwise, the IP is incremented.

The contents of the IP serve as the address for fetching the instruction from the CM. The instruction will be available in the I during phi-3.

$R_{out}\ A_{out} \downarrow$ / $R_{in}\ A_{in} \rightarrow$	00	01	11	10
00	0	0	0	0
01	1	0	0	1
11	1	1	1	1
10	1	0	0	1

R_{out} (next)

$R_{out}\ A_{out} \downarrow$ / $R_{in}\ A_{in} \rightarrow$	00	01	11	10
00	1	1	0	0
01	1	1	1	1
11	1	1	0	0
10	0	0	0	0

A_{out} (next)

Figure 4.3: Karnaugh Map for Handshake module

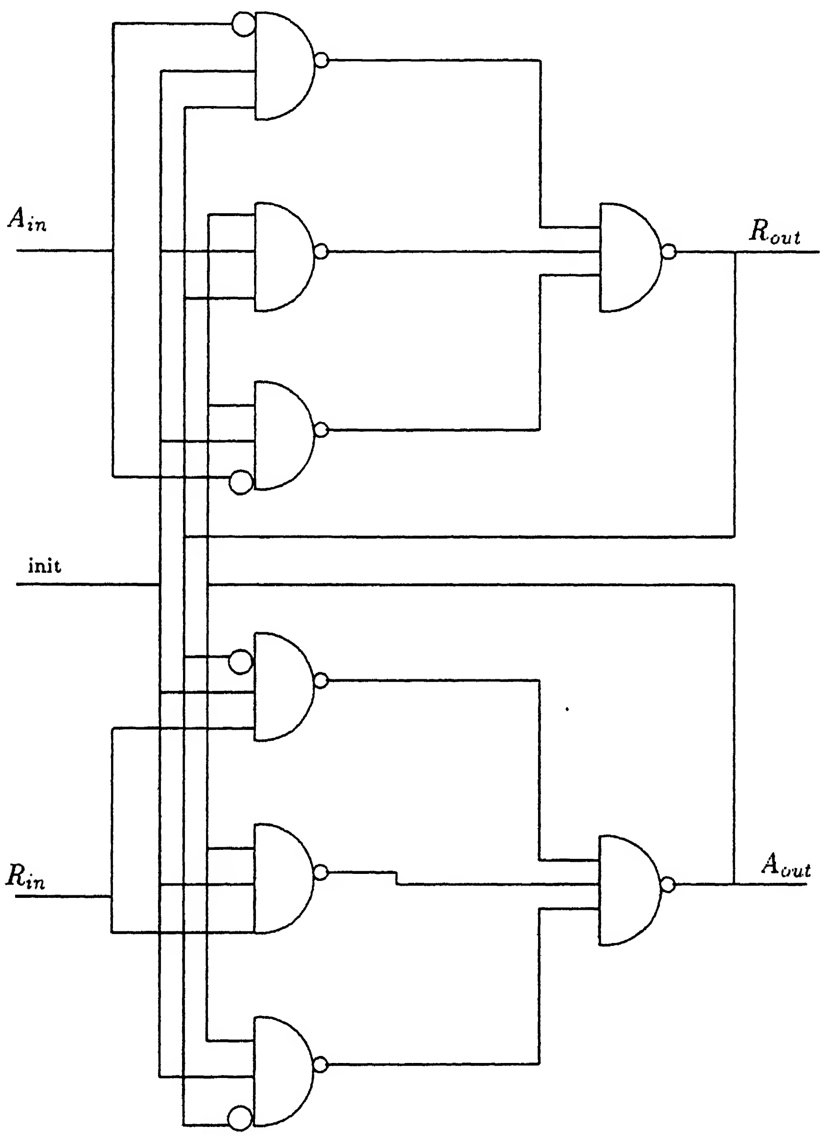


Figure 4.4: Logic Diagram of the Handshake module

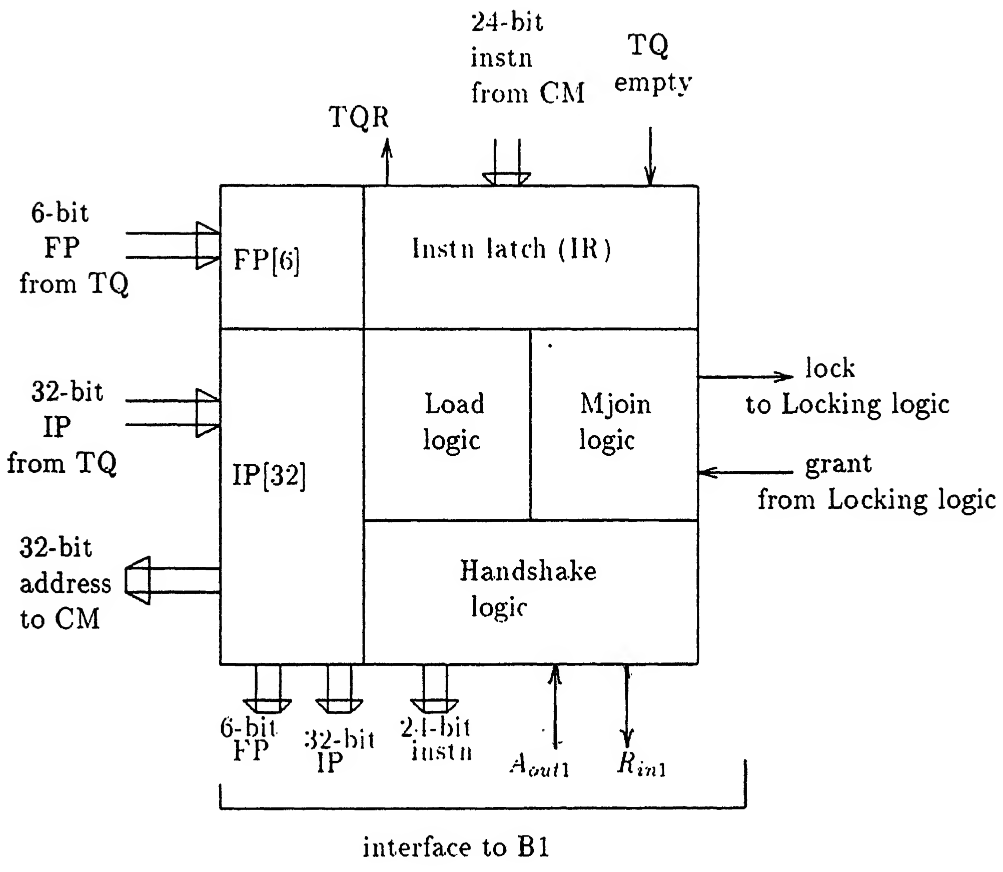
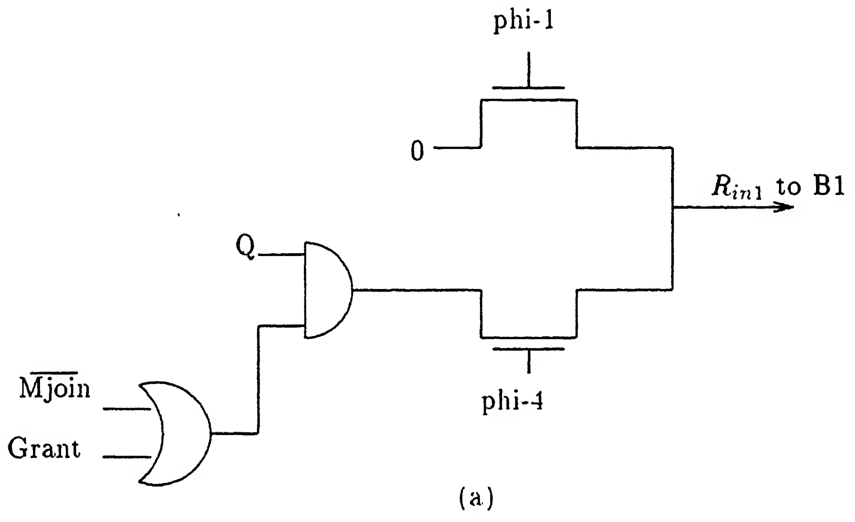


Figure 4.5: Block Diagram of the IFU



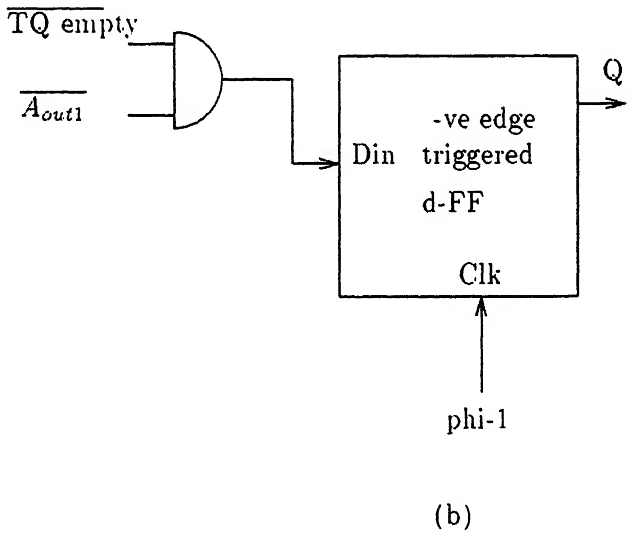


Figure 4.6: Handshaking Logic for IFU

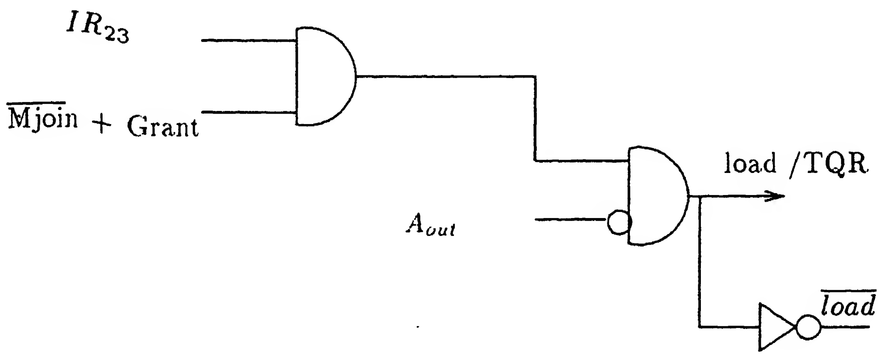


Figure 4.7: Load Logic

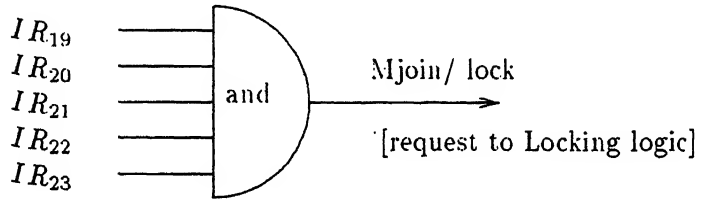


Figure 4.8: Mjoin Logic

The *Mjoin Logic* Fig 4.8 checks if the current instruction is MJOIN and issues a *lock* command to the *Locking Logic*. The *Locking Logic* is common to both the TRSs.

The 24-bit instruction latch(IR) consists of d-latches. Data gets latched in during phi-3.

Fig 4.9 shows the details of the IP[7]. Each bit of the register contains a half-adder[7], a d-latch[7] and a multi plexer[8]. The adder is arranged as an accumulator such that it increments in the same manner as an accumulator. The multiplexer allows the IP to be loaded from the TQ if the *load* line is high.

4.3 The Locking Logic(LL)

The LL resolves the race between the two TRSs to execute MJOIN instruction and issues a *grant* signal to one of the TRSs. It consists of a state tracking +ve edge-triggered T-flipflop(T- ff)[8],two d-FFs and two And gates. Fig 4.10 shows the logic diagram of the LL. Raising of one of the *release* lines

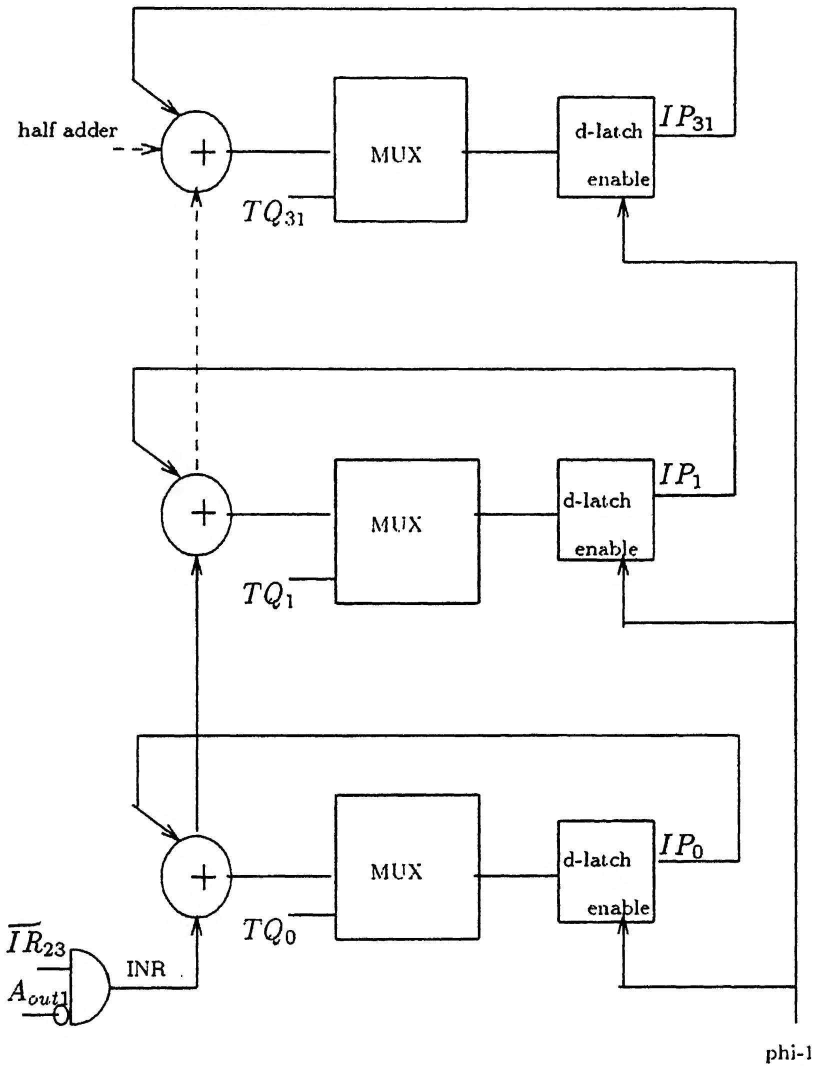
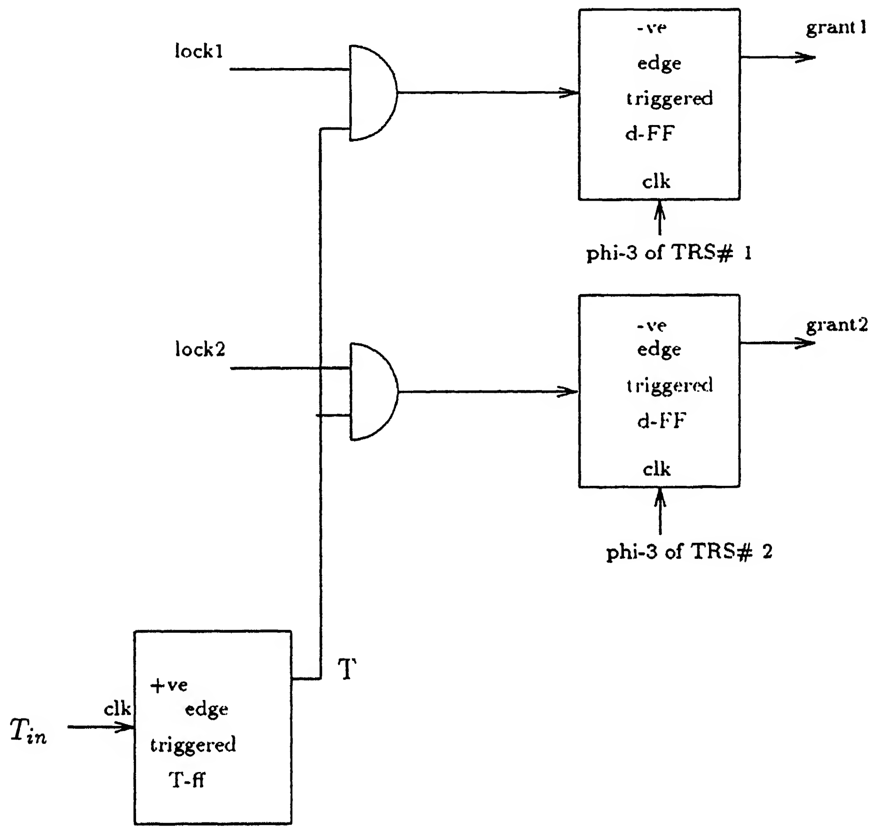


Figure 4.9: Incrementing Register with Load



$T_{in} = grant1 \vee grant2 \vee release1 \vee release2$

Figure 4.10: The Locking Logic

by the RSU of the TRSs sets the t-FF to free(Q=1) state. The *grant* line is set when the Q-output of the t-FF is high and there is a request for *grant*.

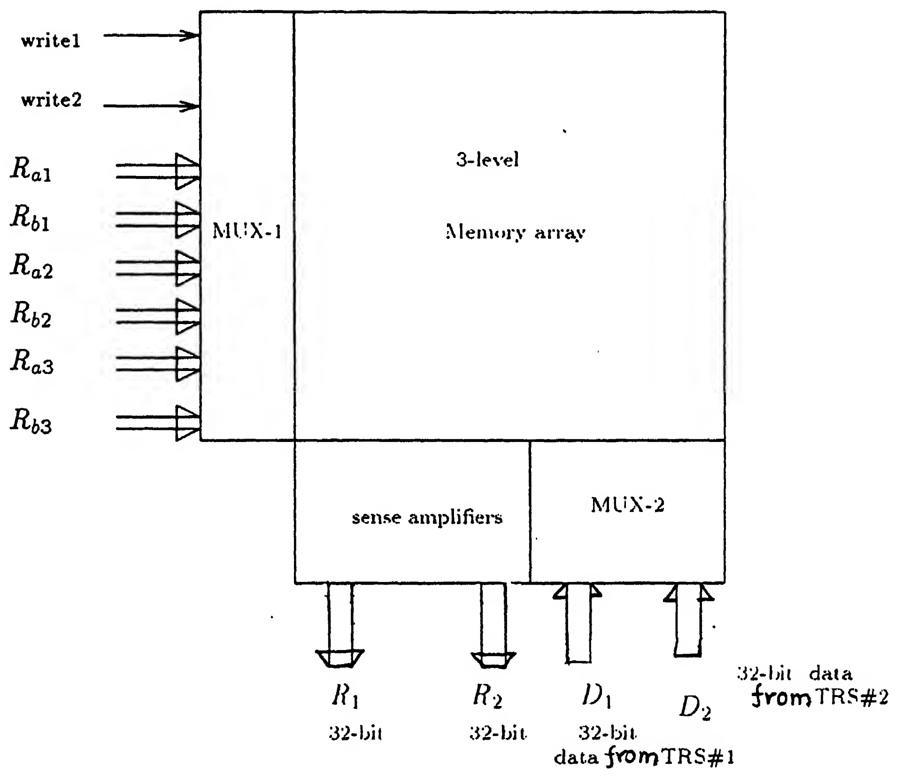
4.4 Operand Memory(OM)

In this section, we discuss the design of the 3-port, 64-word, OM. The OM is organized in a 3-level hierarchical fashion. The block diagram of the OM is shown in Fig 4.11. Two ports are read-only while the third one is a write-only port. The RSU of the TRSs utilize the write port and the OFUs make use of the read ports. The address and data lines of OM is multiplexed such that TRS#1 uses it during phi-1 and TRS#2 uses it during phi-2. The inputs to the OM consists of two 6-bit address lines from the OFU, one address line of 6-bits and a 32-bit data line from the two RSUs. The output lines consist of two 32-bit data lines to the OFU.

Three decoders are necessary for 3-port memories. For simplicity, let us consider only one decoder. Fig 4.12 gives the organization of the such an OM. In level#2, we have two rows of two submodules each, with each submodule having 4 * 4 words. Three row addresses are run vertically along the left side and the 3 column addresses are run horizontally. A_0 is used by the decoder to select one of the two rows of submodules. The select line running through that module is driven. The other two row address lines run horizontally into each of the two rows of submodules, where they serve as column address wires for the submodules. Out of the 3 column address lines, 2 are run vertically in each of the 2 columns of submodules, where they serve as row addresses. The other address line is used by the multiplexer to select one of the two data wires coming out of the columns of the 2 submodules.

Fig 4.12 shows the 3-port static memory MOS cell used in the OM. The circuit consists of two cross-coupled inverters and four pass transistors. The load devices are the P-transistors. To write the cell, DATA is placed on the bit1 line and $\overline{DATA}$ is placed on the $\overline{bit1}$ line. The word line W1 is then asserted. For reading, the corresponding word line is asserted and data will be available on its data line. The sense amplifiers consisting of serially placed ratioed inverters[8] amplifies the output to the required level.

Precharging is done for bit lines of each submodule. Fig 4.14 shows the design of multiplexers used to multiplex the addresses from TRS#1 and TRS#2.



R_{a1} R_{a2} R_{a3} $\longrightarrow$ addresses from TRS #1

R_{b1} R_{a2} R_{b3} $\rightarrow$ addresses from TRS #2

Figure 4.11: The Operand Memory

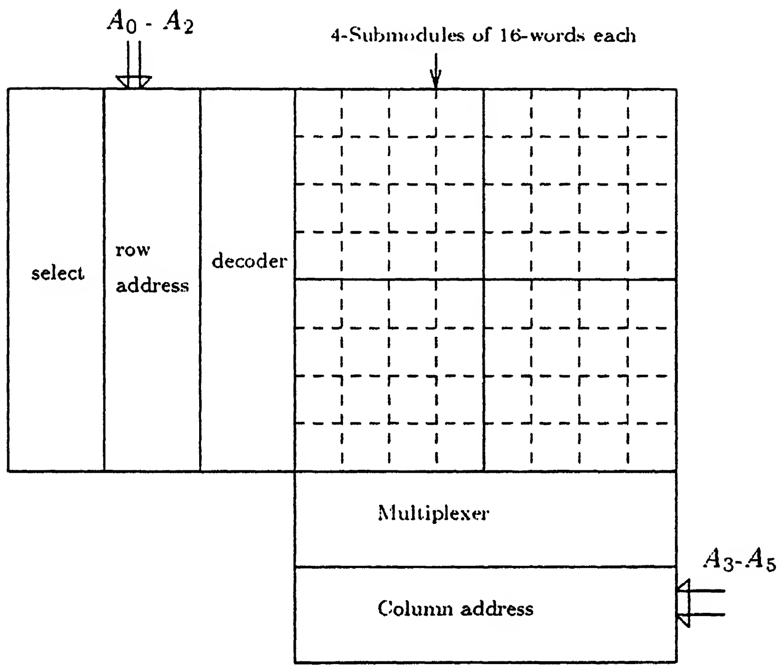


Figure 4.12: Level#2 of the Operand Memory

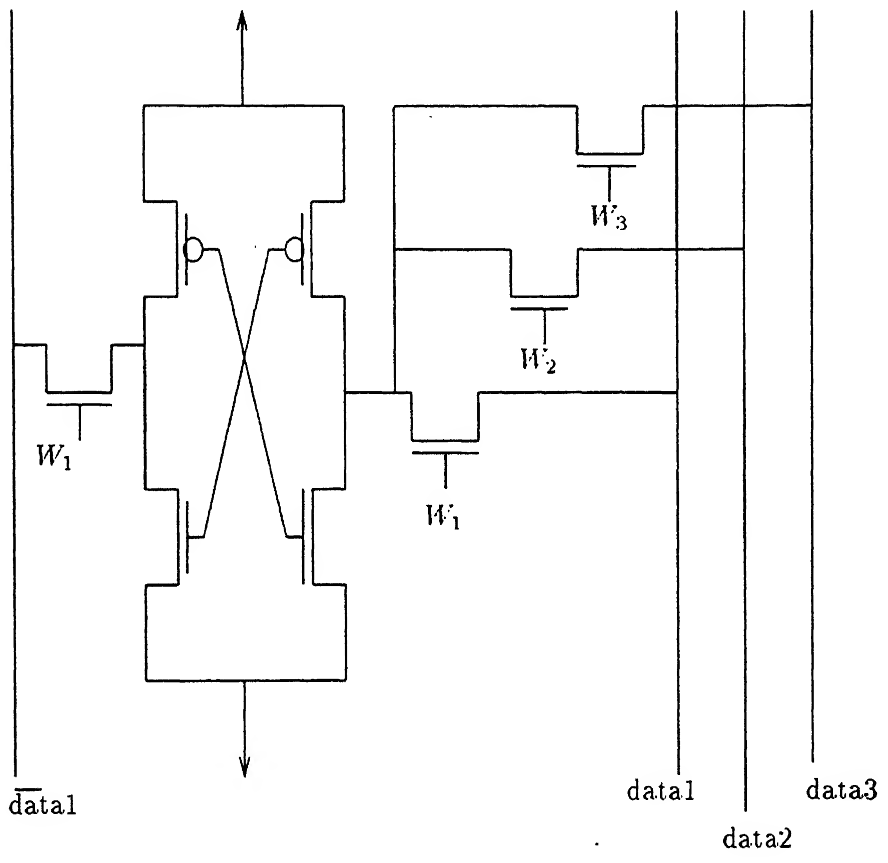


Figure 4.13: The Memory Cell

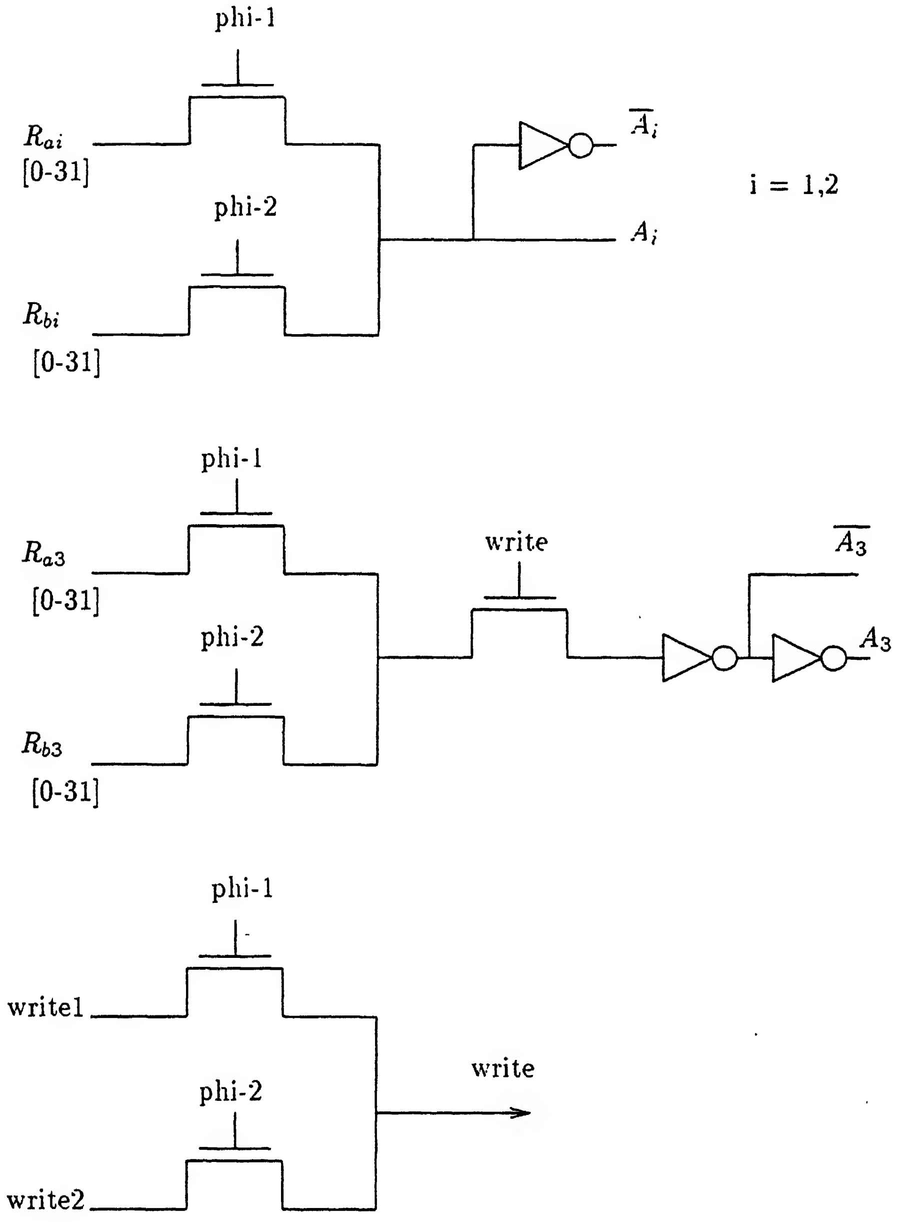


Figure 4.14: Multiplexer of Operand Memory

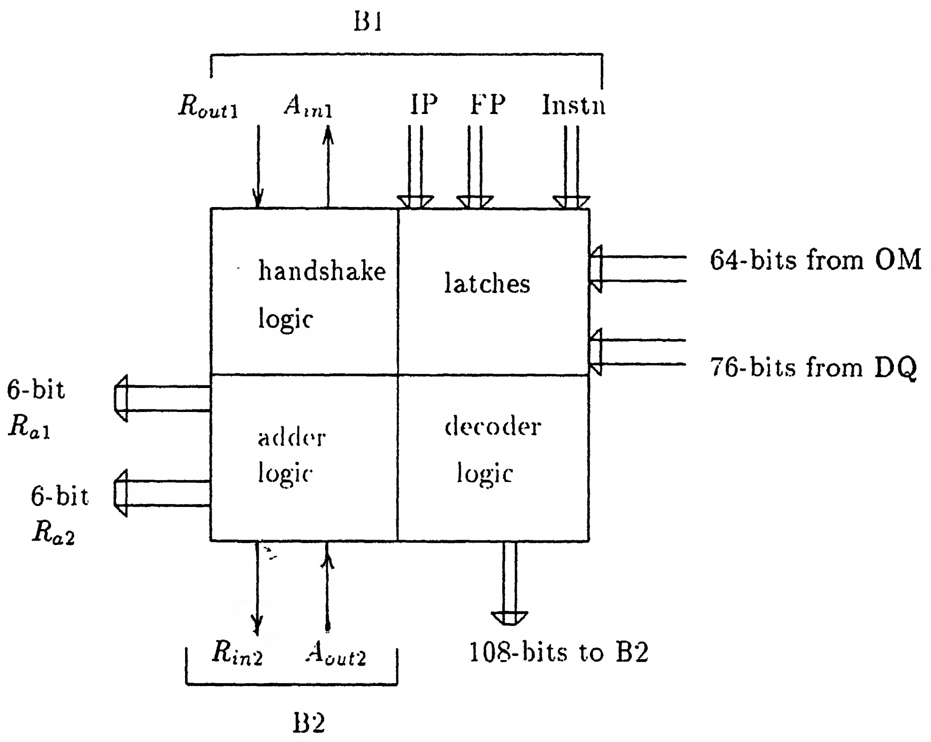


Figure 4.15: Block Diagram of the OFU

4.5 Operand Fetch Unit(OFU)

The main function of the OFU module is to fetch operands from the OM. If the instruction being executed is RESM, then operands are fetched from the DQ. The block diagram showing the inputs and outputs of the OFU is given in Fig 4.15. 62-bits of data consisting of FP(6-bits), IP(32-bits) and instruction(24 bits) are accepted from the buffer *B1*. The DQ returns 76-bits consisting of 32-bits of data, 6-bit address of the destination register(DR) and 38-bits of CT(6-bit FP,32-bit IP). The input lines from the OM consist of two operands, each of 32-bit length. Two handshaking signals(one from *B1* and one from *B2*) also come to the OFU. The output lines consist of 114 lines(6-bit instruction, two 32-bit operands, a 6-bit destination register(DR), 32-bit IP, 6-bit FP) to the buffer *B2* between OFU and the EXU, two 6-bit addresses to the OM, one DQ request line and two handshaking lines, one each to *B1* and *B2*.

During phi-1, handshaking signal R_{in2} is made low. If A_{out2} goes low indicating that the buffer *B2* can accommodate incoming data at the end of the current cycle. operations of the OFU can proceed normally. A_{in1} is made high if R_{out1} is high, requesting data from *B1*. The data is now latched into the 62-bit d-latch(L1). The d-FF, triggered by the negative edge of phi-1, keeps track of the state of execution of the OFU(Q=1 means normal operation). During phi-3, R_{in2} is made high depending on the state of the d-FF. Fig 4.16 shows the details of the handshaking logic for the OFU.

The Decoder logic makes use of IR_{18} and IR_{19} of the instruction latch(IR) to determine whether the instruction is of type *no fetch*, *one fetch*, *two fetch* or *DQ fetch*. Four 2-bit decoders are used to for this purpose. Fig 4.17 gives the details of the Decoder logic.

Two 6-bit Binary Lookahead Carry (BLC)[7] adders are used to add the FP with two 6-bit OM addresses (bits 6-11 and 12-17 of the instruction latch(IL)) to obtain FP+r1 and FP+r2. The resultant addresses are sent to the OM for fetching the operands. Fig 4.18 gives the details of the BLC adder.

During phi-2, if *DQ fetch* line is high, a request to DQ is made through the DQfetch line. At the end of phi-2, data from the DQ(if DQfetch line is high), or the operands from the OM are ready. The incoming data from the OM is stored in latches R1 and R2. Data from the DQ is latched into FP, IP, destination register(DR) and R1. The details of multiplexing the inputs for these latches is shown in (Fig 4.19).

The output of R2 is multiplexed with IR before sending it to *B2*. Fig 4.20 illustrates the details of the multiplexer.

4.6 The Execution Unit(EXU)

The operation of the EXU is similar to that of the conventional ALU. Fig 4.21 illustrates the input/output interfaces and the block diagram of the EXU. 114 input lines (6-bit instruction, 32-bit operand R1, 32-bit operand R2, 6-bit DR, 32- bit IP. 6 bit FP) come from the buffer *B2*. Three handshaking lines, one each from *B2*, *B3* (situated between EXU and RSU) and *B4*(situated between EXU and The sequencer) come into the EXU. The output lines consist of 45 lines to the buffer *B3*, 38 lines to *B4* and 78 lines to the message processor(MP).

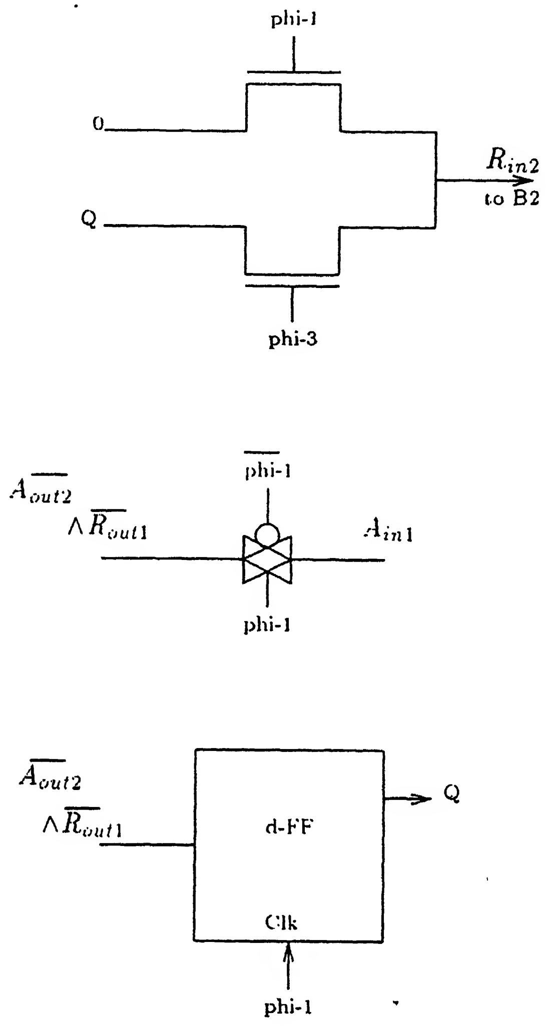


Figure 4.16: Handshaking Logic for OFU

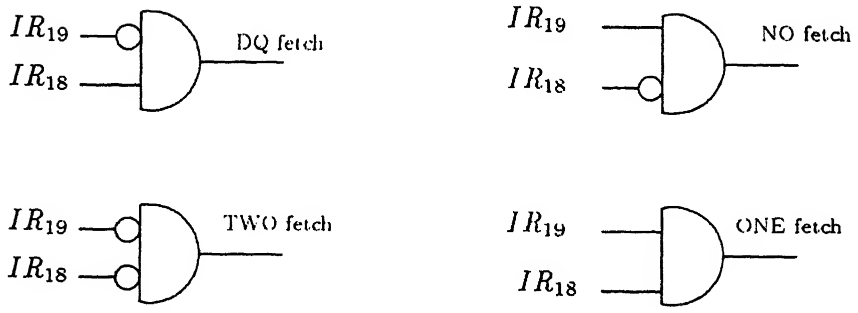


Figure 4.17: Decoder Block for OFU

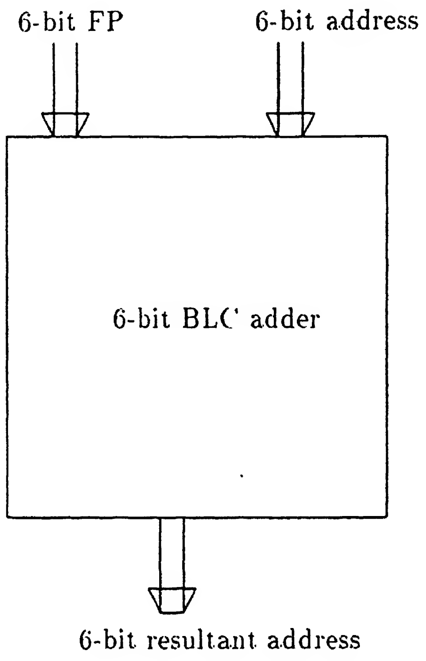


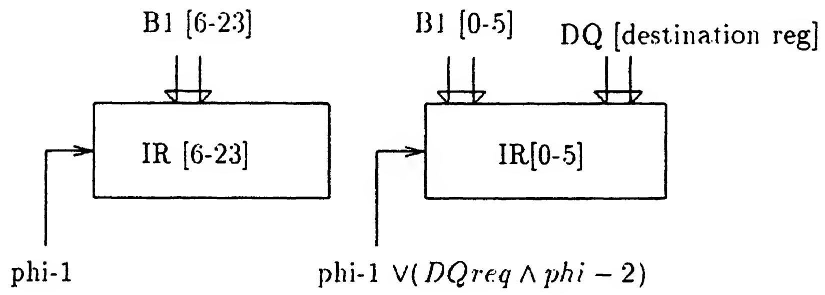
Figure 4.18: The BLC Adder

During phi-1, handshaking signals R_{in3} and R_{in4} are made low. If A_{out3} and A_{out4} go low, indicating that the buffers *B3* and *B4* can accommodate incoming data at the end of the current cycle, the EXU can operate normally. A_{in2} is now made high if R_{out1} is high and the contents of the downcounting register (of MJOIN logic) is zero. The d- ff(state tracking d flip-flop) is set if normal operation is being performed in the current cycle (Fig 4.22) and the 114-bit input from *B2* is latched into latches INST(6-bit), R1(32-bit), R2(32-bit), DR(6-bit) and FP(6-bit) respectively.

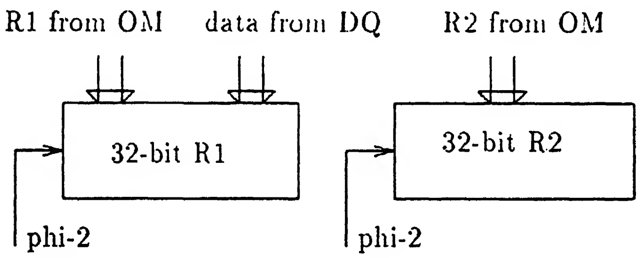
The decoding circuitry consists of 4-bit decoders for the instructions ADD, SUB, MJOIN, JZ, JP, JPZ, JNZ, LOAD, JUMP, CHFP. The instructions RESM, JMP, LOADX require 3-bit decoders and the instructions AND, OR, XOR, STORE, SFTL, SFTR, MFORK, STOREX need 5-bit decoders.

The Jump Logic (Fig 4.23) takes the contents of R2 as the input and determines whether the conditions Positive(P), positive or Zero(PZ), Zero(Z) or Negative or Zero(NZ) are true. The lines P, PZ, Z, NZ turn high if its respective logic is true.

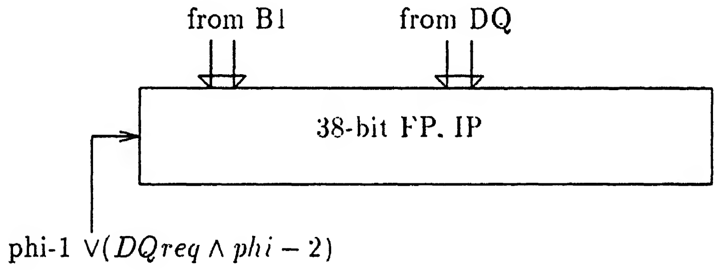
The Mfork Logic (Fig 4.24) consists of a 3-bit downcounting regis-



(a) Instruction latch inputs



(b) Multiplexed inputs for R1 and R2



(c) Multiplexed inputs for FP, IP

Figure 4.19: Multiplexed Inputs for Latches

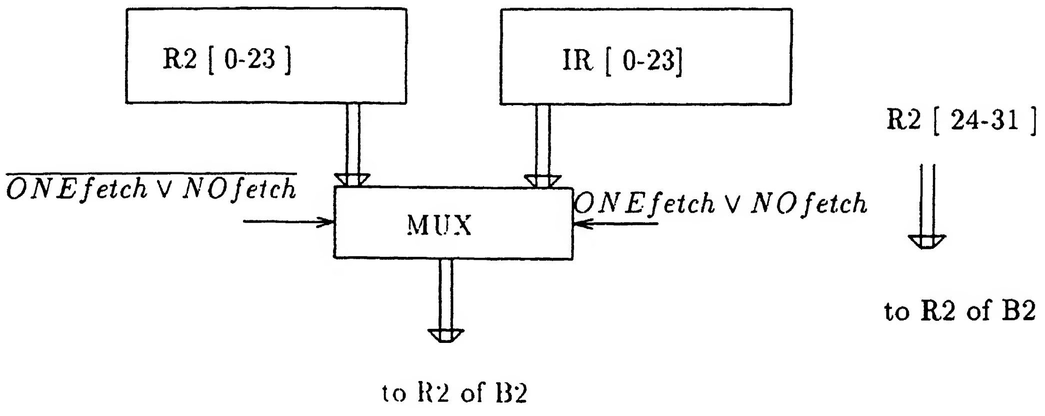


Figure 4.20: Multiplexed output for R2

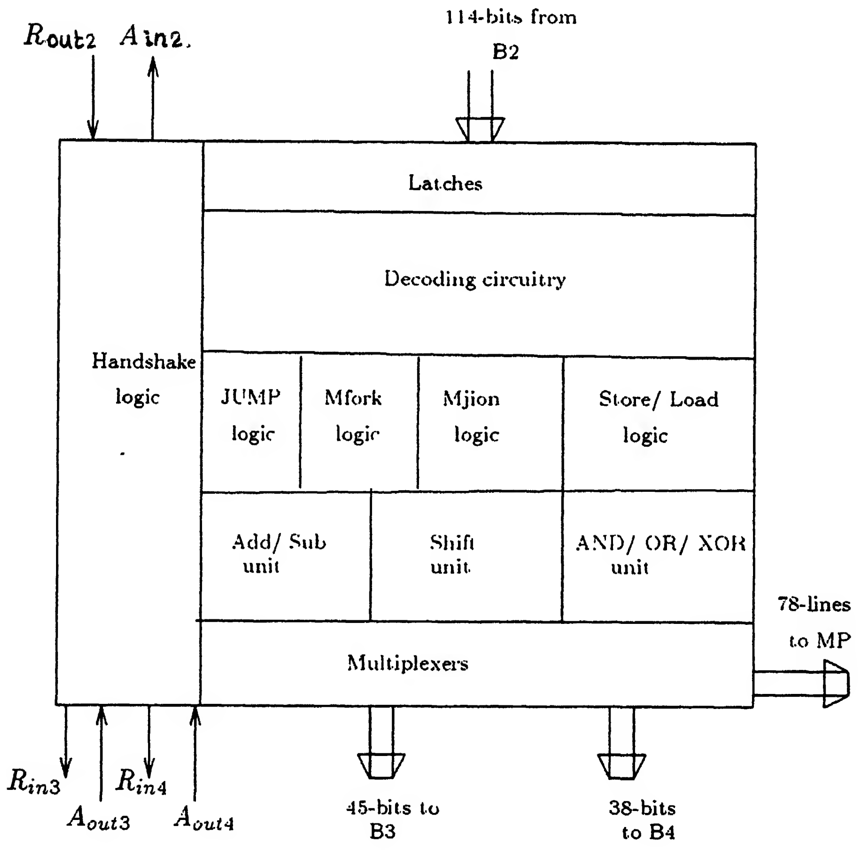


Figure 4.21: Block Diagram of the EXU

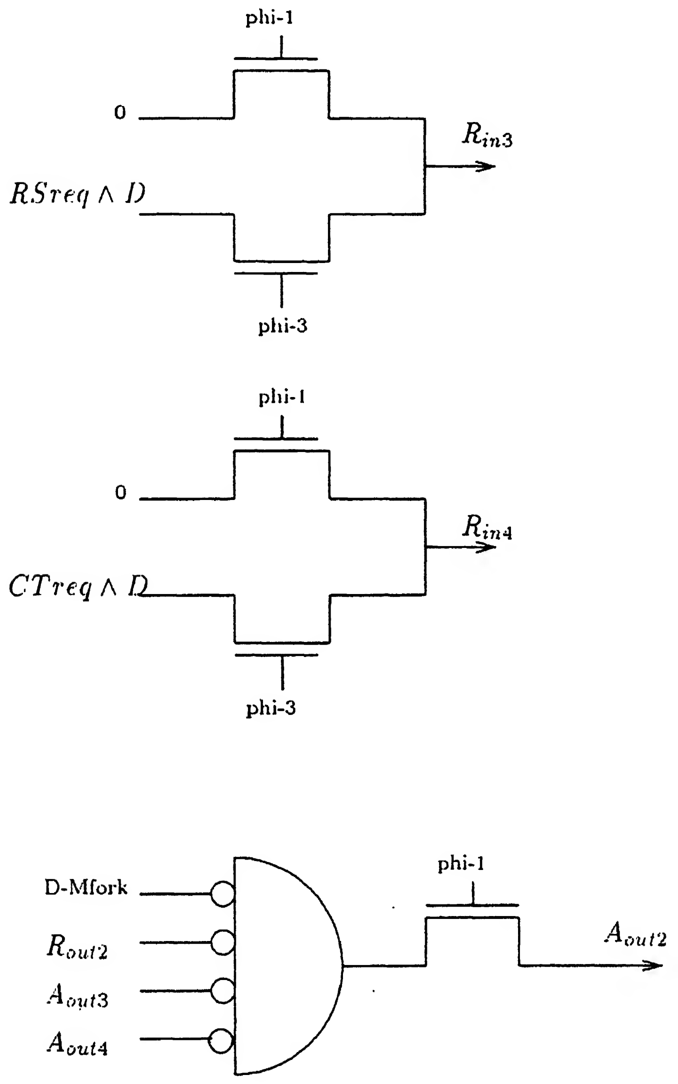


Figure 4.22: Handshaking Circuitry for EXU

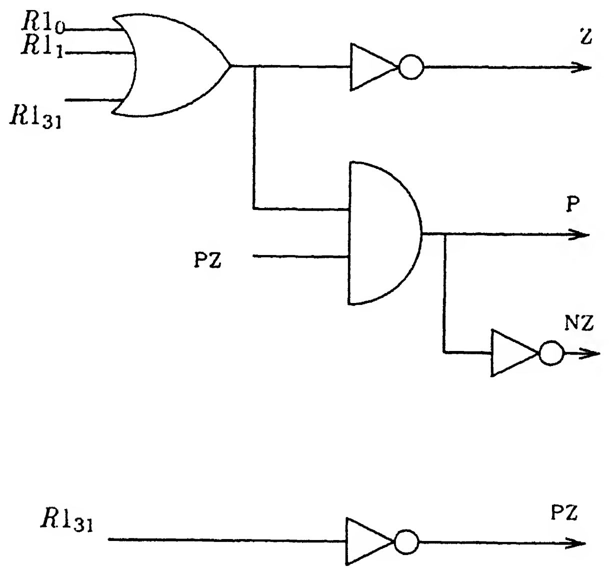


Figure 4.23: The Jump Logic

ter(DCR) with load option, decoder circuitry and Multiplexer Logic. The downcounting register is loaded when a MFORK instruction is encountered. The loading circuitry determines whether 0, 1, 2, 3 or 4 is to be loaded into the DCR .The contents of the DCR is decremented in the subsequent clock cycles till it becomes zero. New instructions are not taken up by the EXU until the contents of the DCR is zero. The decoder circuitry determines whether the contents of the DCR is 0, 1, 2, 3 or 4 and the multiplexer correspondingly directs the contents of R1 to the Add/Sub unit.

The Mjoin Logic (Fig 4.25) consists of a 3-input And gate to determine if the content of R1 is zero. If so, a new Continuation token is generated and sent to the CTU.

The Store/Load unit (Fig 4.26) sends a proper message to the Message Processor. The multiplexing circuitry prepares 78 bit package consisting of one bit Read/Write signal, 32-bit address a, 32-bit IP and 6-bit FP along with one bit request line.

The Add/Sub unit consists of a set of eight 4-bit Manchester Carry Lookahead[7] adders to make a 32-bit adder. Less than 1K transistors are required for a 32-bit Manchester Carry Lookahead adder. The Manchester chain takes more time to propagate the carry than in a parallel adder, but time is not a crucial factor for the EXU. Fig 4.27 gives the block diagram

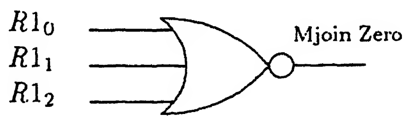


Figure 4.25: The Mjoin Logic

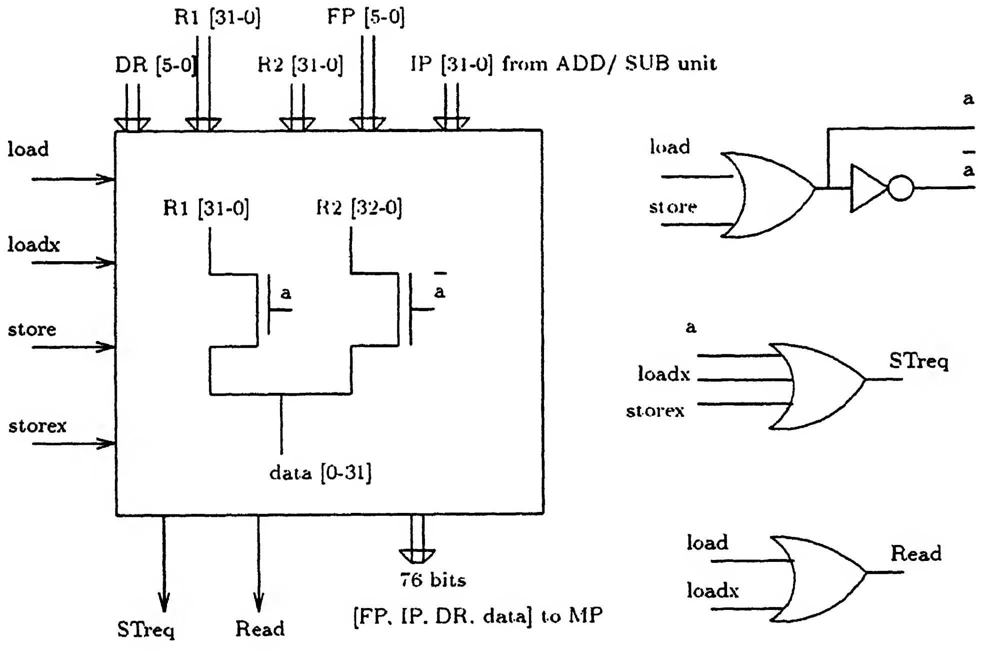


Figure 4.26: The Store/Load Unit

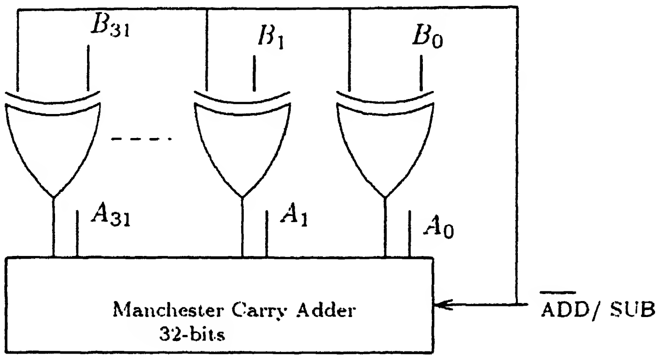


Figure 4.27: The Add/Sub Unit

of the adder unit.

The Add/Sub unit is used to add/sub data as well as for the generation of the Continuation tokens for jump/memory type of instructions. The multiplexed input circuitry for the INP1 and INP2 of Add/Sub unit is shown in Fig 4.28.

A 32-bit *barrel shifter* [7] is used to implement the Shift unit. The details of the OR/XOR/AND unit are given in Fig 4.29.

The Multiplexing unit directs the outputs of the Add/Sub unit, Shift unit, OR/XOR/AND unit to the buffers *B3* and *B4*. Fig 4.30 gives the details of the output for B3 and Fig 4.31 gives the details of the multiplexed outputs for FP and IP to B4.

4.7 The Result Store Unit(RSU)

The main function of the RSU is to store the result generated by the EXU in the OM, the address to which is pointed by the contents of the Destination Register(DR). 45 inputs come to the RSU from *B2*. Fig 4.32 illustrates the block diagram of the RSU. The output of the RSU goes to the TQ.

The RSU first checks if data is present in the buffer *B3* ($R_{out3} = 0$) and raises A_{in3} during phi-1. The contents of FP is added to the the contents of DR using a 6-bit Manchester Carry lookahead adder and the resultant address is placed on the address lines going to the OM. The 32-bit data is placed on the data lines.

Bit-0 of the incoming data from *B3* indicates that the current instruction is MJOIN ($b_0 = 1$). This line is sent to the Locking Logic during phi-1.

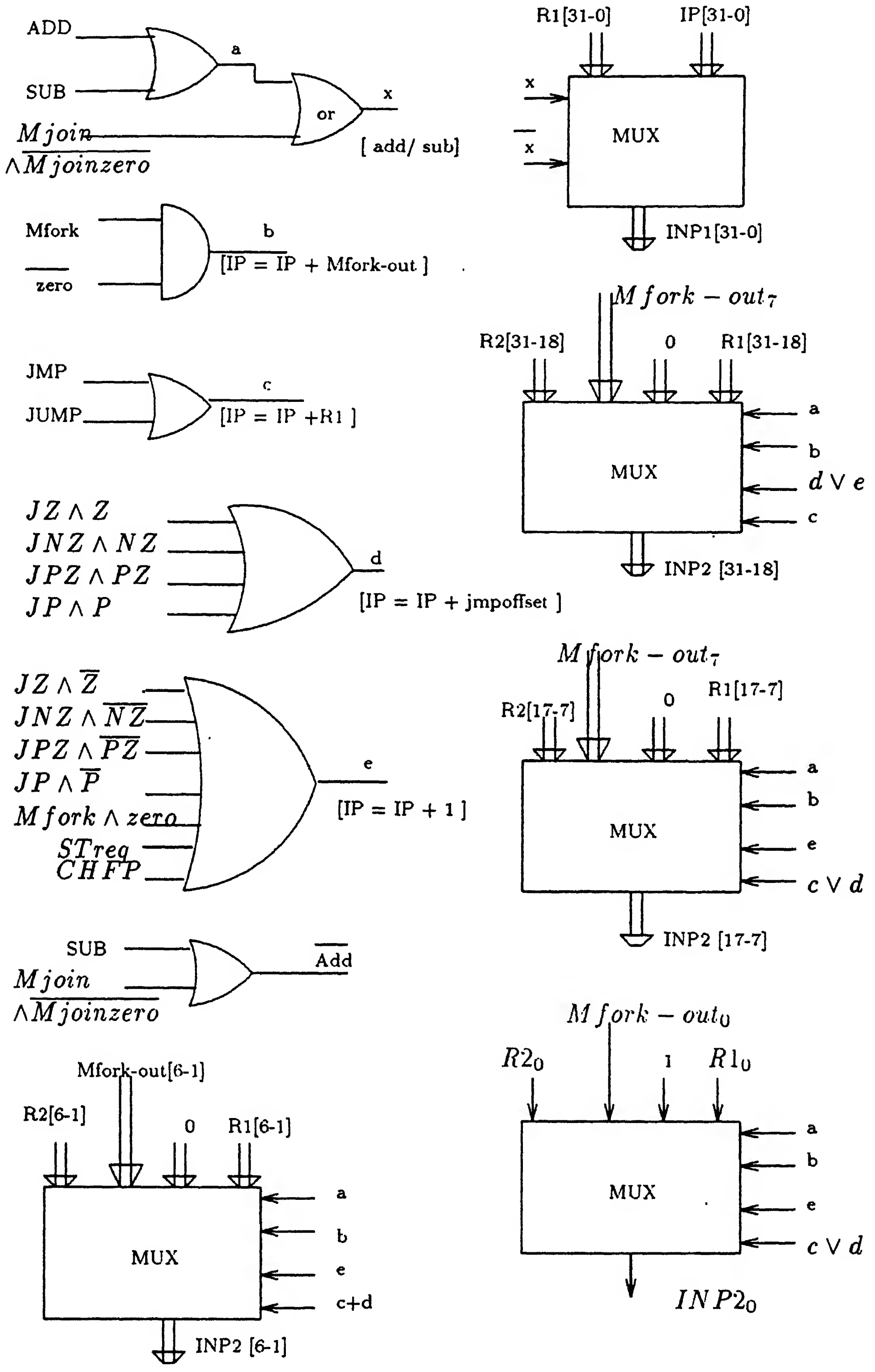


Figure 4.28: The inputs to the Add/Sub Unit

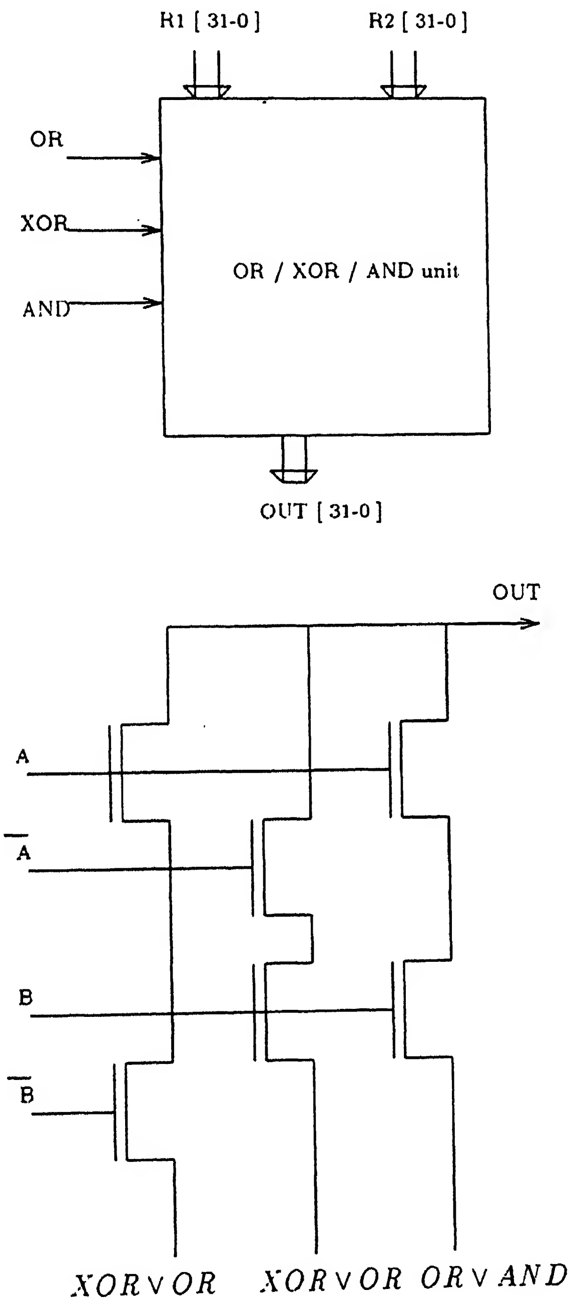


Figure 4.29: The OR/XOR/AND Unit

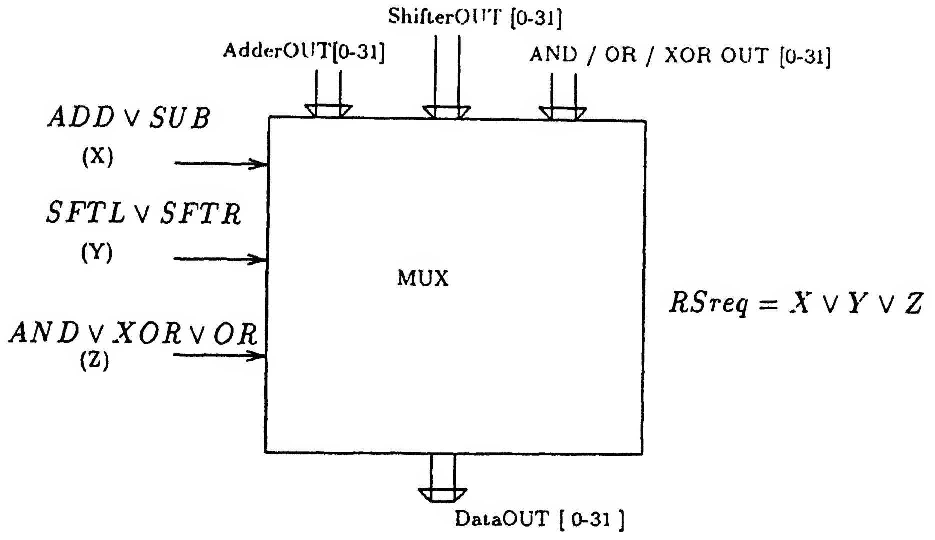
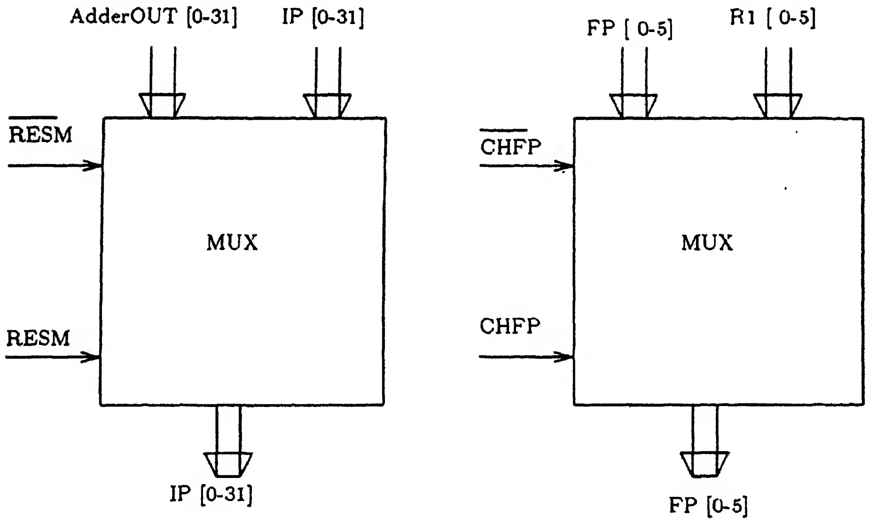


Figure 4.30: Outputs for B3



$$CTreq = IR_{23} \wedge (\overline{Mjoin} \vee Mjoinzero) \wedge \overline{Read}$$

Figure 4.31: Outputs for B4

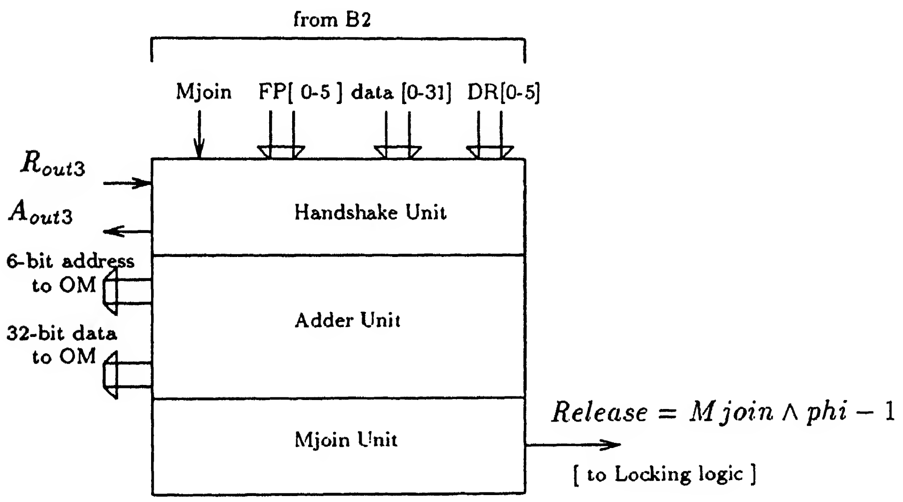


Figure 4.32: Block Diagram of the RSU

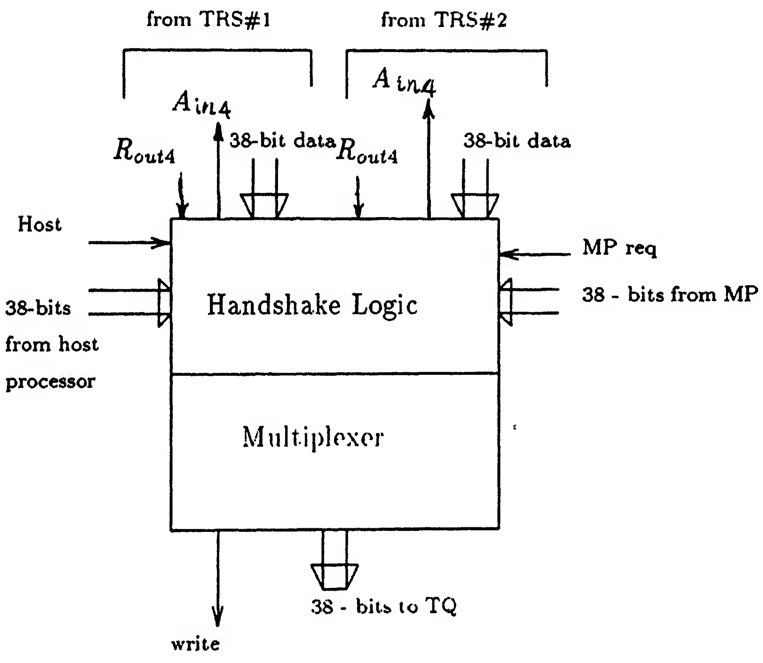
4.8 The Sequencer

The Sequencer is outside the TRS. It simply consists of a multiplexer which directs the Continuation Tokens(CT) from TRS#1, TRS#2,the Host processor and the Message Processor(MP) to the TQ. CT from TRS#1 is routed to TQ during Phi-1, TRS#2 during phi-2 , MP during phi-3 and Host processor during phi-4. Fig 4.33 gives the details of the Sequencer.

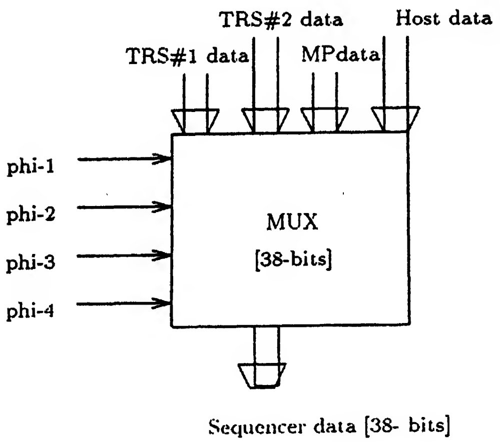
4.9 The Token Queue(TQ) and the Data Queue(DQ)

The TQ is a FIFO(First In First Out) which stores Continuation Tokens. It can be initially loaded by the Host Processor through the Sequencer. The block diagram of the TQ is given in Fig 4.34. The inputs to the TQ come from the Sequencer. The TQ supplies tokens to the IFU of the two TRSs.

The TQ consists of six words of 38-bits. Each queuecell is made up of three pass transistors(A, B and C) and two inverters. A d- ff is associated with each word. It keeps track of the state of each queue word(Q_c = 1 implies word contains a CT). Fig 4.35 gives the structure of a single *queuecell* and the associated state circuitry. Q_n indicates the state of the next queue word and Q_p that of the previous word. Read (R) and Write(W) operations can be performed simultaneously during a single cycle. The Queue makes use of two phase clocking(phi-1 and phi-2). For the first word, Q_p = 0 and Q_n = 1 for the last word. During *write* operation (W=1), all cells with



(a) The Block Diagram



(b) The Multiplexer

Figure 4.33: Block Diagram of the Sequencer

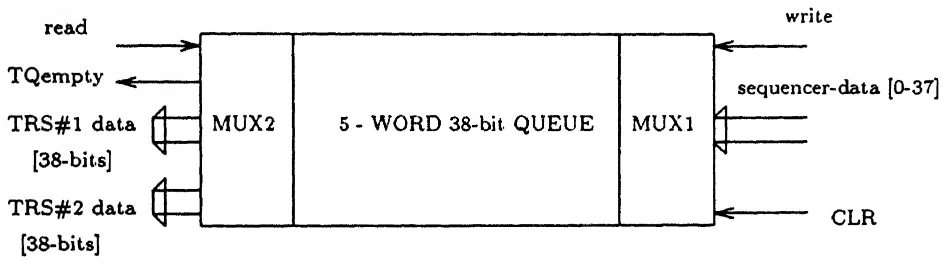


Figure 4.34: Block Diagram of the TQ

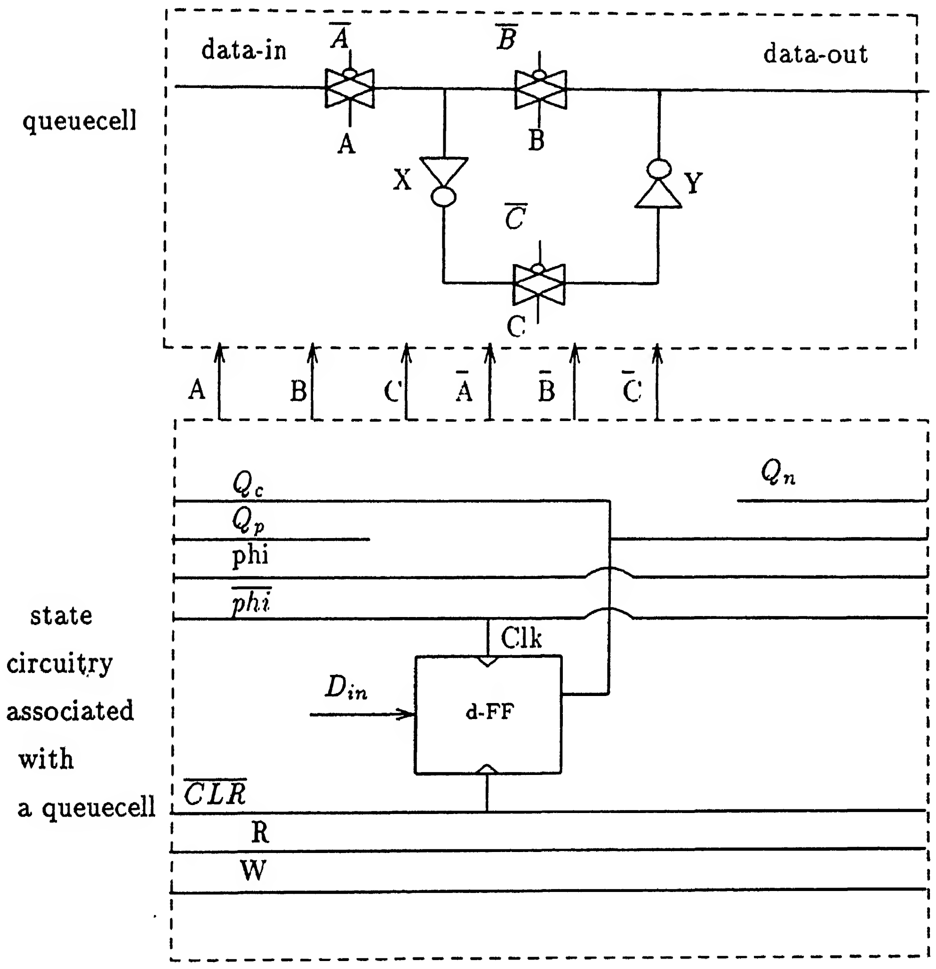
its state as well as that of the succeeding cell as empty (Q_c = 0, Q_n = 0), are *transparent*. In these cells, pass transistors A and C are *on* during phi-1. A cell with Q_c = 0 and Q_n = 1 is *blocking* and only A is *on* during phi-1. In cells which are neither *transparent* nor *blocking*, pass transistor B is *on* during phi-1. During *read*, all cells with Q_c = 1 and Q_p = 1 copy the data from the previous cell. Pass transistor A is made *on* for such cells during phi-1. Pass transistor B will be *on* for all other cells during phi-1. Irrespective of the operation, C is *on* during phi-2 for all the cells. When *read* and *write* operations are simultaneously required, the cell with Q_c = 0 and Q_n = 1 will additionally have pass transistor C to be *on*.

The state diagram for Q_c of the d-FF is given in Fig 4.36.

Fig 4.37 shows the block diagram of the DQ giving the details about the inputs and the outputs. The basic *queuecell* is as in Fig 4.35.

4.10 Message Processor

The MP is the interface between the global I-structure controller and the *Twine-RISC*. It can be divided into two parts, viz., the *sending unit* and the *receiving unit*. The *sending unit* simply multiplexes the output of the Store/Load units of the two EXUs and sends the 78-bit (1-bit Read/Write, 32-bit address *a*, 32-bit IP, 6-bit FP, 6-bit DR, 1-bit request line) read/write request to the I-structure controller. The *receiving unit* receives a packet of 77-bits(1-bit request, 32-bit data, 6-bit DR, 6-bit FP, 32-bit IP) and sends it to the DQ. It also generates a token (0,0) and sends it to the TQ if there is a request from the I-structure controller. Fig 4.38 gives the details of the MP.



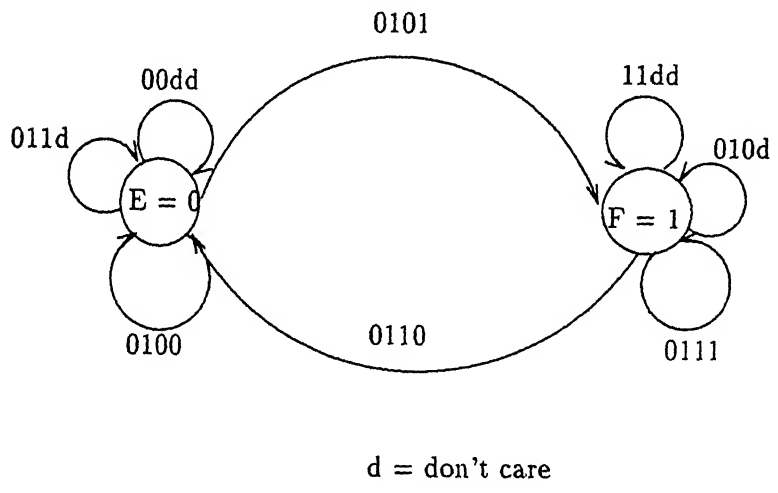
$$A = ((W \wedge \overline{Q_c}) \vee (R \wedge Q_c)) \wedge phi$$

$$B = \overline{((W \wedge \overline{Q_c}) \vee (R \wedge Q_c))} \wedge phi$$

$$C = ((W \wedge \overline{Q_c} \wedge \overline{Q_n}) \vee (R \wedge \overline{Q_c} \wedge Q_n) \wedge phi) \vee \overline{phi}$$

$$D_{in} = ((Q_n \wedge W \wedge \overline{R}) \vee (Q_p \wedge R \wedge \overline{W}) \vee (Q_c((W \wedge \overline{R}) \vee (\overline{W} \wedge R)))$$

Figure 4.35: The Queuecell



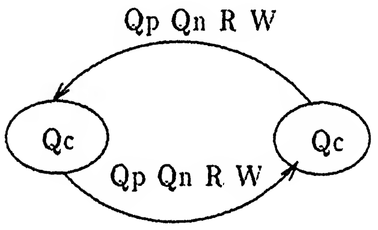


Figure 4.36: The State Diagram for D-ff

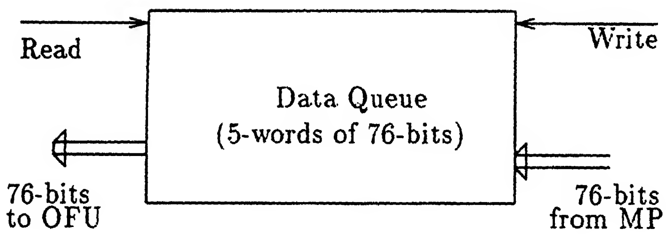
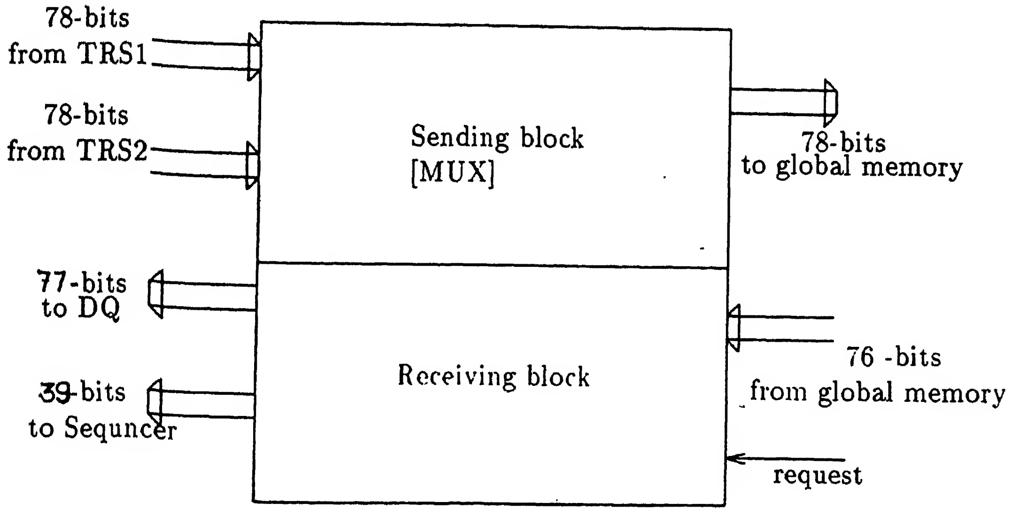
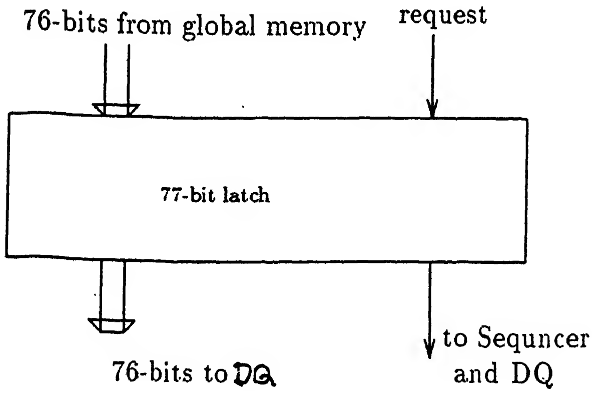


Figure 4.37: The Block Diagram of DQ



(a) The Block diagram



(b) Receiving Unit

Figure 4.38: The Message Processor

4.11 Conclusion

In this chapter we have described the hardware realization of *Twine-RISC*. The IFU requires approximately 700 transiters, the Locking Logic 100, The Operand Memory(64-words) 18500, the OFU 1750, the TQ(5-words of 38-bits) 1700, the DQ(8-words of 76-bits) 5600, the RSU 1000, the Sequencer 200 and the EXU 3000. Thus, the number of transistors required for a two stream *Twine-RISC* is less than 40K and hence it can be put in a single chip. The above realization can be implemented in VLSI either in Psuedo-NMOS or in C-MOS technology. In the next chapter, we conclude the thesis with results and a brief note on the future work to be done in this field.

Chapter 5

Conclusion

Our main aim in this thesis has been to propose a simple hardware design for *Twine-RISC.* In the design proposed, less than 40k transistors are required for a 2-stream *Twine-RISC.* We have also implemented the novel design of the *queuecell* (explained in the previous chapter) in VLSI and excellent results have been obtained from this exercise.

5.1 Features of *Twine-RISC* design

- The design allows the easy expansion of *Twine-RISC* in terms of number of TRSs.
- Handshaking units are present between each pair of TRS blocks thus aiding asynchronous operation and implementing semi-self timed circuits.
- Each TQ and DQ operation(read or write) takes less than 1ns. The operation of the queue is very fast and it can easily cope up with more TRSs in the *Twine-RISC.*
- A hierarchical design has been proposed for the OM. This reduces the delay while reading the data from the memory.
- Loading and incrementing the Instruction Pointer are done in parallel in the IFU , thereby reducing the operation time of the IFU, which is a major bottleneck.
- Decoding of instruction is distributed across the units to simplify the decoding logic and make it fast.

- MJOIN instruction is made mutually exclusive to avoid possible deadlock in the system.

5.2 Deficiencies

- Though two TRSs bring out the fine parallelism exploitation concept, they still remain to effectively exploit the fine-grain parallelism since a typical program will have a fine-grain parallelism of degree much more than 2. With a change in technology, it should be possible to include more than 2 TRSs.
- A lot of extra software support is needed for managing the OM. However, this can be done by the compiler at compile time.
- While increasing the number of TRSs, more ports should be provided in the OM, making it more complex.
- The design supports a very small TQ and DQ. However it is very easy to increase the size by replicating the VLSI layout. For a realistic 256 word TQ and 128-word DQ sites, 2-stream *Twine-RISC* will require approximately 200k transistors.

5.3 Extensions and future work

- VLSI implementation of our proposed design should be done after determining the optimum size of TQ and DQ through software simulation. The size of the above mentioned queues depend on the specific applications.
- Appropriate clocking circuitry should be provided for the design. The blocks of the TRSs have 3-4 phases and the operation of each block should be properly timed.
- The number of TRSs per *Twine-RISC* should be increased to at least 4. This will help in exploiting fine-grained parallelism more effectively.
- Software support like the assembler, loader etc should also be provided. The software requirement of *Twine-RISC* is different from that of existing machines. For example, the compiler should identify the

parallelism in the program and code it using MFORK and MJOIN instructions.

Appendix A

Instruction Set for Twine-RISC

The instruction set of *Twine-RISC* is a simple extension of RISC models. Out of the 21 instructions, 20 instructions can be executed in a single cycle. Only MFORK requires more than one cycle. Instructions are classified into four major categories viz. Arithmetic and Logic, Branch, Memory reference and Generation and synchronization of multiple threads. In this appendix, we describe the instruction set of *Twine-RISC*.

A.1 Arithmetic and Logic group

This group of instructions perform the arithmetic and logic instructions

A.1.1 ADD, SUB, AND, OR, XOR instructions

These instructions fetch two operands from the OM and store one result in the OM.

The syntax of these instructions is

opcode r1, r2, r3 where

r1 - left operand source register.

r2 - right operand source register.

r3 - destination register.

The OFU fetches two operands [FP + r1] and [FP + r2]. EXU performs the required operation. The resultant value r3 is passed on to RSU. RSU stores the value in [FP + r3]. Execution continues from the next location (IP + 1) and no continuation token is generated.

A.1.2 SFTL, SFTR instructions

Operand can be shifted left or right by 32-bits. The shift count is stored in the instruction in 6-bits (IR_{12} - IR_{17}). One operand is fetched from the OM and the result is stored in the destination register.

The syntax is

opcode a, r2, r3.

a - number of bits to be shifted.

r2 - operand source register.

r3 - destination register.

The OFU fetches one operand [FP + r2]. The EXU shifts the operand by the required number of bits and passes the value and r3 to RSU. RSU stores the value in [FP + r3]. The execution continues from IP + 1.

A.2 Branch instructions

These are the jump group of instructions.

A.2.1 JMP instruction

JMP supports direct jump up to a address range of 2^{18}-1. There is no operand fetch.

The syntax is

JMP x

x - 18-bit value specifying the jump address.

x is treated as an operand. EXU generates continuation token (FP. IP + x) and passes it to the Sequencer which forwards it to the TQ.

A.2.2 JUMP instruction

JUMP supports jump to a address range of 2^{32}-1. There is a single operand fetch.

The syntax is

JUMP r1.

r1 - register specifying the jump offset.

The OFU fetches one operand [FP + r1]. EXU generates a continuation token (FP. IP + [r1]) and passes it to the Sequencer which forwards it to the TQ.

A.2.3 JZ, JP, JPZ, JNZ instructions

These instructions support conditional jump up to 12-bit offset from the current IP. As jump offset is directly specified in IR_5 - IR_{17} of the instruction, only one operand is fetched from the OM (where the condition is stored).

The syntax for these instructions is

JCND r1, x

r1 - condition operand source register.

x- 12-bit value specifying the jump offset.

OFU fetches one operand [FP + r1] from the OM. EXU tests for the condition in this register. If the condition is true, the EXU generates a continuation (FP. IP + [r1]). Otherwise, (FP, IP + 1) is generated and inserted into the TQ through the Sequencer.

A.3 Special instructions

These instructions are extensions to the conventional RISC instruction set.

A.3.1 MFORK instruction

MFORK spawns parallel threads of computation from an executing thread. Four possible new threads are organized as 8-bit offsets n1, n2, n3, n4 and grouped into a 32-bit word. This is stored in an OM location. One operand fetch is required for this instruction.

The syntax is

MFORK r1, r3

r1 - operand source register from which new thread offsets are derived.

r3 - destination register.

OFU fetches an operand [FP + r2] from OM. The EXU interprets it as four bytes of 8-bits each. For each byte, if the value is non-zero, a new thread is generated. EXU prepares a continuation token of (FP, IP + offset value) for each non-zero offset value and sends it to the Sequencer. One continuation (FP, IP + 1) is always generated. EXU passes (number of threads, r3) to the RSU. RSU stores the number of threads in [FP + r3].

A.3.2 MJOIN instruction

The MJOIN instruction helps in the synchronization of multiple threads. This instruction decrements the specified register content by 1 and writes the result back in the same location.

The syntax of MJOIN is

MJOIN rl, rl.

rl - operand source register which contains the number of threads to be synchronized.

OFU fetches one operand [FP + r2] from the OM. The EXU decrements it by 1 if the value is not zero. If the value is zero, the continuation (FP, IP + 1) is passed to the EXU. The RSU stores the value in the register (FP + r2).

A.3.3 CHFP instruction

This instruction changes the FP with the first 6-bits of the specified register.

The syntax is CHFP rl.

rl - operand source register.

The OFU fetches [FP + rl] from the OM. The EXU changes the contents of the FP by the first 6-bits of [FP + rl] and sends the resultant continuation to the TQ through the Sequencer.

A.4 Memory based instructions

These instructions are used to move data to and from the global memory.

A.4.1 LOAD instruction

This instruction is used to load data from the global memory to the OM.

The syntax of this instruction is

LOAD rl, r3.

rl - operand source register containing the global memory location.

r3 - destination register in the OM.

OFU fetches one operand [FP + rl] from OM and gets the global memory location. EXU sends a read request to MP which then forwards it to the memory controller of the global memory. The request format is (read, STreq, [FP + rl], FP, IP + 1, r3). STreq indicates that there is a request from the TRS to the global memory controller. After the instruction is completed by the memory controller, it responds by sending (Req, v, FP, IP + 1, r3) to the MP. Req indicates that memory controller is sending data and v is the 32-bit data value. MP inserts the value (v, FP, IP + 1, r3) in the DQ and sends a continuation (0, 0) to the TQ through the Sequencer.

A.4.2 LOADX instruction

The syntax is

LOADX a, r3.

a - 6-bit value specifying the address of the global memory location in the instruction.

r3 - destination register.

OFU does not fetch any operand from the OM. The message format to the global memory controller is (read, STreq, a, FP, IP + 1, r3). After the instruction is completed by the memory controller, it responds by sending (Req, v, FP, IP + 1, r3) to the MP. Req indicates that memory controller is sending data and v is the 32-bit data value. MP inserts the value (v, FP, IP + 1, r3) in the DQ and sends a continuation (0, 0) to the TQ through the Sequencer.

A.4.3 RESM instruction

This instruction is executed to move the data from the DQ to the OM. Operands are fetched from the DQ.

The syntax is

RESM

OFU fetches data(v, FP, IP + 1, r3) from DQ instead of OM. EXU passes (v, r3) to the RSU. EXU also prepares a continuation (FP, IP + 1) which is then forwarded to the TQ through the Sequencer. RSU stores v in [FP + r3].

A.4.4 STORE instruction

This instruction is used to store data into the global, memory from the OM.

The syntax is

STORE r1, r2.

r1 - operand source register containing the address of the global memory location.

Appendix B

Table of instructions

	Instrn name	6-bit opcode	descripton about the other 18-bits
1	ADD	000000	r1, r2, r3: three 6-bit addresses;
2	SUB	000100	r1, r2, r3: three 6-bit addresses;
3	AND	010000	r1, r2, r3: three 6-bit addresses;
4	OR	010100	r1, r2, r3: three 6-bit addresses;
5	MJOIN	111111	r1, r1: 6-bit r1[17-12], 6-bit r1[5-0];
6	XOR	011000	r1, r2, r3: three 6-bit addresses;
7	SFTL	010011	a., r2, r3: 6-bit shift number, two 6-bit addresses;
8	SFTR	010111	a., r2, r3: 6-bit shift number, two 6-bit addresses;
9	MFORK	011011	r1, r3:6-bit r1[17-12], 6-bit r3[5-0];
10	STOREX	011111	r1, x : 6-bit r1[17-12], 6-bit x[5-0];
11	STORE	011100	r1, r2 : two 6-bit addresses;
12	LOADX	110110	a, r3: 6- bit address a, 6-bit r3;
13	LOAD	111011	r1, r3:6-bit r1[17-12], 6-bit r3[5-0];
14	JUMP	110011	r1 : 6-bit register specifying offset;
15	JMP	110010	x : 18-bit value specifying jump address;
16	JZ	100011	r1, x : 6-bit r1, 12-bit jump offset x;
17	JP	100111	r1, x : 6-bit r1, 12-bit jump offset x;
18	JNZ	101111	r1, x : 6-bit r1, 12-bit jump offset x;
19	JPZ	101011	r1, x : 6-bit r1, 12-bit jump offset x;
20	RESM	111101	other bits are not required;
21	CHFP	110111	r1 : 6-bit register giving new FP;

Figure B.1: Instruction Format

Bibliography

[1] R. Nikhil, Arvind. *Can Dataflow Subsume Von Neumann computing?*, Proc. 16th Int. Symp. On Computer Architecture, Jerusalem, Israel, June 1989. pp 262-272.

[2] R.Moona, S.Nandy, V.Rajaraman. *Twine RISC: An Architecture for Simultaneous Execution of Multiple Threads.* [Personal Communications].

[3] David A. Patterson and Carlo H. Sequin, *A VLSI RISC*, IEEE Computer, Sept. 1982, pp.8-21.

[4] Arvind, D. Culler. *Dataflow Architectures*, Annual Reviews in Computer Science, Vol 1, Annual Reviews Inc., Palo Alto, CA, 1986. pp 225-253.

[5] Arvind, Rishiyur S. Nikhil and Keshav K. Pingali, *I-Structures : Data structures for parallel computing*, Proc. Workshop on Graph Reduction, Los Alamos NM, Sept-Oct. 1986.

[6] Arvind, Rishiyur S. Nikhil. *Executing a Program on the MIT Tagged-Token Dataflow Architecture*, IEEE *Trans. Comp.*. 1989.

[7] Weste, N. H. E. and Eshraghian, K. (1985). *Principles of CMOS VLSI design: A system perspective Approach.* Addison-Wesely Publishing Company.

[8] Mead, C. A and Conway, L. (1980). *Introduction to VLSI Systems.* Addison-Wesely, Reading, Mass.

[9] Arvind, S. Brobst. *The Evolution of Dataflow Architectures from Static Dataflow to P-RISC.* Technical Report, CSG Memo 316, MIT Lab for Computer Science, August 1990.

[10] Arvind, D. Culler, Ianucci, V. Kathail, K. Pingali and R.Thomas. *The Tagged Token Dataflow Architecture*. Technical Report, MIT Lab for Computer Science, October 1984.

[11] D. Culler, G. Papadopoulos. *The Explicit Token store*, CSG Memo 312, MIT Lab for Computer Science, June 1990.

[12] G. Papadopoulos, K. Traub. *Multithreading : an Explicit Token Store Architecture*, Proc. 17th Int. Symp. on Computer Architecture, Seattle, Washington, May 1990. pp 82-91.